O'REILLY®

7-Е ИЗДАНИЕ

Рассматривается .NET Standard 2

# C# 7.0 Справочник

## ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА

**Джозеф Албахари и Бен Албахари**

# C# 7.0

## Справочник

*ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА*

# C# 7.0

# IN A NUTSHELL

*Joseph Albahari and Ben Albahari*

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

# C# 7.0

# Справочник

*ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА*

*Джозеф Албахари и Бен Албахари*

Диалектика

Москва · Санкт-Петербург
2018

ББК 32.973.26-018.2.75
А45
УДК 681.3.07

Компьютерное издательство "Диалектика"

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

**Албахари**, Джозеф, **Албахари**, Бен.

А45 C# 7.0. Справочник. Полное описание языка. : Пер. с англ. – СпБ. : ООО "Альфа-книга", 2018. – 1024 с. : ил. – Парал. тит. англ.

ISBN 978-5-6040043-7-1 (рус.)

**ББК 32.973.26-018.2.75**



*Научно-популярное издание*

**Джозеф Албахари, Бен Албахари**

# C# 7.0. Справочник
# Полное описание языка

Подписано в печать 02.04.2018. Формат 70×100/16
Гарнитура Times. Печать офсетная.
Усл. печ. л. 82,56. Уч.-изд. л. 67,4
Тираж 300 экз. Заказ № 3233

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит А, пом. 848

ISBN 978-5-6040043-7-1 (рус.)

ISBN 978-1-491-98765-0 (англ.)

# Оглавление

# Содержание

# Об авторах

**Джозеф Албахари** – автор книг *C# 6.0 in a Nutshell* (*C# 6.0. Справочник. Полное описание языка*, ИД "Вильямс", 2016 год), *C# 6.0 Pocket Reference* (*C# 6.0. Карманный справочник*, ИД "Вильямс", 2016 год), *C# 7.0 Pocket Reference* (*C# 7.0. Карманный справочник*, "Диалектика", 2017 год) и *LINQ Pocket Reference*. Он также является создателем LINQPad (`http://www.linqpad.net`) – популярной утилиты для подготовки кода и проверки запросов LINQ.

**Бен Албахари** – соучредитель веб-сайта Auditionist, предназначенного для кастинга актеров в Соединенном Королевстве. На протяжении пяти лет он занимал должность руководителя проектов в Microsoft и работал над несколькими проектами, включая .NET Compact Framework и ADO.NET.

Он был соучредителем Genamics, поставщика инструментов для программистов на C# и J++, а также программного обеспечения для анализа цепочек ДНК. Бен выступал в качестве соавтора *C# Essentials*, первой книги по языку C#, выпущенной издательством O'Reilly, и предшествующих изданий *C# in a Nutshell*.

# Об иллюстрации на обложке

Животное, изображенное на обложке книги – это нумидийский журавль. За грацию и гармоничность нумидийский журавль (лат. *Antropoides virgo*) также называют журавль-красавка. Данный вид журавля считается местным для Европы и Азии; на зимний период его представители мигрируют в Индию, Пакистан и северо-восточную Африку.

Хотя нумидийские журавли являются самыми маленькими среди семейства журавлиных, они защищают свои территории так же агрессивно, как и другие виды журавлей, громкими голосами предупреждая других особей о нарушении границы. При необходимости они вступают в бой. Нумидийские журавли гнездятся не в болотистой местности, а на возвышенностях, и могут даже жить в пустынях при наличии воды на расстоянии от 200 до 500 метров. Временами для кладки яиц они строят гнезда, окружая их мелкими камешками, но чаще яйца откладываются прямо на землю, будучи защищенными только растительностью.

В некоторых странах нумидийские журавли считаются символом удачи, а иногда даже защищаются законом.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой исчезновения; все они важны для нашего мира. Чтобы узнать больше о том, чем вы можете помочь, посетите веб-сайт `animals.oreilly.com`.

Изображение на обложке воспроизводит оригинальную гравюру XIX века.

# Благодарности

## Джозеф Албахари

Прежде всего, я хочу поблагодарить своего брата Бена Албахари за то, что он уговорил меня взяться за книгу *C# 3.0 in a Nutshell*, успех которой и привел к появлению последующих изданий. Бен разделяет мою готовность ставить здравомыслящие вопросы и упорство в разборе сложных вещей на более простые, пока не станет ясно, как все *действительно* работает.

Я был весьма польщен иметь дело со столь выдающимися техническими рецензентами. В этом и предыдущем изданиях мы располагали бесценными и всесторонними отзывами от Рода Стивенса, Джареда Парсонса, Стивена Тауба, Метью Грува, Диксина Яна, Ли Коварда, Бонни Девитт, Вонсок Чхэ, Лори Лалонд и Джеймса Монтеманьо.

Данная книга построена на основе предшествующих изданий, чьих технических рецензентов я также безмерно уважаю: Эрика Липперта, Джона Скита, Стивена Тауба, Николаса Палдино, Криса Барроуза, Шона Фаркаса, Брайана Грюнкмейера, Маони Стивенс, Девида Де Винтера, Майка Барнетта, Мелитты Андерсен, Митча Вита, Брайана Пика, Кшиштофа Цвалины, Мэтта Уоррена, Джоэля Побара, Глин Гриффитс, Иона Василиана, Брэда Абрамса, Сэма Джинтайла и Адама Натана.

Я высоко ценю тот факт, что многие технические рецензенты являются состоявшимися личностями в компании Microsoft, и особенно благодарен им за то, что они уделили время, способствуя переходу настоящей книги на новый качественный уровень.

Наконец, я хочу поблагодарить команду O'Reilly, в том числе моего лучшего редактора Брайана Макдональда, и персонально Ли, Мири и Соню.

## Бен Албахари

Поскольку мой брат написал свои благодарности первым, я охотно присоединяюсь к ним. Мы начали программировать, будучи еще детьми (в те времена мы делили между собой один Apple IIe; брат писал собственную операционную систему, а я занимался написанием игры Hangman), поэтому очень здорово, что мы сейчас пишем книги вместе. Я надеюсь, что полезный опыт, который мы обрели во время написания данной книги, выльется в полезные сведения, которые вы почерпнете, читая ее.

Я также хотел бы поблагодарить моих бывших коллег из Microsoft. Там работает много умных людей, не только в плане интеллекта, но также и в более широком эмоциональном смысле, и я скучаю по работе с ними. В частности, я многому научился у Брайана Бекмана, которому весьма обязан.

# Предисловие

Язык C# 7.0 представляет собой шестое крупное обновление флагманского языка программирования от Microsoft, позиционирующее C# как язык с невероятной гибкостью и широтой применения. С одной стороны, он предлагает высокоуровневые абстракции, подобные выражениям запросов и асинхронным продолжениям, а с другой стороны, обеспечивает низкоуровневую эффективность через конструкции вроде специальных типов значений и необязательное использование указателей.

Платой за развитие становится относительно трудное освоение языка. Несмотря на то что такие инструменты, как Microsoft IntelliSense (и онлайновые справочники), великолепно помогают в работе, они предполагают наличие концептуальных знаний. Настоящая книга предлагает такие знания в сжатой и унифицированной форме, не утомляя беспорядочными и длинными введениями.

Подобно предшествующим четырем изданиям книга организована вокруг концепций и сценариев использования, что делает ее пригодной как для последовательного чтения, так и для просмотра в произвольном порядке. Хотя допускается наличие только базовых навыков, материал рассматривается довольно глубоко, что делает книгу ориентированной на читателей средней и высокой квалификации.

В книге раскрываются язык C#, среда CLR и основные сборки .NET Framework. Мы решили сконцентрироваться на этом для того, чтобы затронуть такие сложные темы, как параллелизм, безопасность и домены приложений, без ущерба для глубины или читабельности. Функциональные возможности, появившиеся в C# 6, C# 7 и связанной с языком инфраструктуре .NET Framework, отмечены особым образом, так что настоящую книгу можно применять также и в качестве справочника по версиям C# 5 и C# 6.

## Предполагаемая читательская аудитория

Книга рассчитана на читателей средней и высокой квалификации. Предварительное знание языка C# не обязательно, но необходимо наличие общего опыта программирования. Для начинающих данная книга будет дополнять, но не заменять вводный учебник по программированию.

Эта книга является идеальным дополнением к любой из огромного множества книг, ориентированных на прикладные технологии, такие как ASP.NET, WPF, UWP или WCF. В книгах подобного рода языку и инфраструктуре .NET Framework обычно уделяется минимальное внимание, тогда как в данной книге все это рассматривается подробно (и наоборот).

Если вы ищете книгу, в которой кратко описаны все технологии .NET Framework, то настоящая книга не для вас. Она также не подойдет, если вам нужно изучать API-интерфейсы, которые специфичны для разработки приложений, ориентированных на мобильные устройства.

# Как организована эта книга

В трех главах, следующих сразу после вводной, внимание сосредоточено целиком на языке C#, начиная с описания основ синтаксиса, типов и переменных и заканчивая такими сложными темами, как небезопасный код и директивы препроцессора. Новички должны читать указанные главы последовательно.

В остальных главах рассматривается платформа .NET Framework, в том числе следующие темы: LINQ, XML, коллекции, параллелизм, ввод-вывод и работа в сети, управление памятью, рефлексия, динамическое программирование, атрибуты, безопасность, домены приложений и собственная способность к взаимодействию. Большинство этих глав можно читать в произвольном порядке кроме глав 6 и 7, которые закладывают фундамент для последующих тем. Три главы, посвященные LINQ, также лучше читать последовательно, а в некоторых главах предполагается наличие общих знаний параллелизма, который раскрывается в главе 14.

# Что требуется для работы с этой книгой

Примеры, приводимые в книге, требуют наличия компилятора C# 7.0 и инфраструктуры Microsoft .NET Framework 4.6/4.7. Также полезно иметь под рукой документацию .NET от Microsoft, чтобы просматривать справочную информацию по отдельным типам и членам (документация доступна в онлайновом режиме).

Хотя исходный код можно писать в простом редакторе вроде Блокнота и запускать компилятор в командной строке, намного продуктивнее работать с *инструментом подготовки кода* для немедленного тестирования фрагментов кода и с *интегрированной средой разработки* (integrated development environment – IDE) для построения исполняемых файлов и библиотек.

В качестве инструмента подготовки кода загрузите LINQPad 5 из веб-сайта `http://www.linqpad.net` (бесплатно). Утилита LINQPad полностью поддерживает C# 7.0 и сопровождается одним из авторов книги.

В качестве IDE-среды загрузите Microsoft Visual Studio 2017: для целей книги подойдет любая редакция кроме бесплатной версии Express.

Все листинги кода для глав 2–10 плюс глав, посвященных параллелизму, параллельному программированию и динамическому программированию, доступны в виде интерактивных (редактируемых) примеров LINQPad. Для загрузки примеров перейдите на вкладку Samples (Примеры) в окне LINQPad, щелкните на ссылке Download/import more samples (Загрузить/импортировать дополнительные примеры) и в открывшемся окне щелкните на ссылке Download C# 7.0 in a Nutshell samples (Загрузить примеры для книги *C# 7.0 in a Nutshell*).

# Соглашения, используемые в этой книге

Для иллюстрации отношений между типами в книге применяется базовая система обозначений UML (рис. 0.1). Параллелограммом обозначается абстрактный класс, а окружностью – интерфейс. С помощью линии с незакрашенным треугольником обозначается наследование, причем треугольник указывает на базовый тип. Линия со стрелкой определяет однонаправленную ассоциацию, а линия без стрелки – двунаправленную ассоциацию.

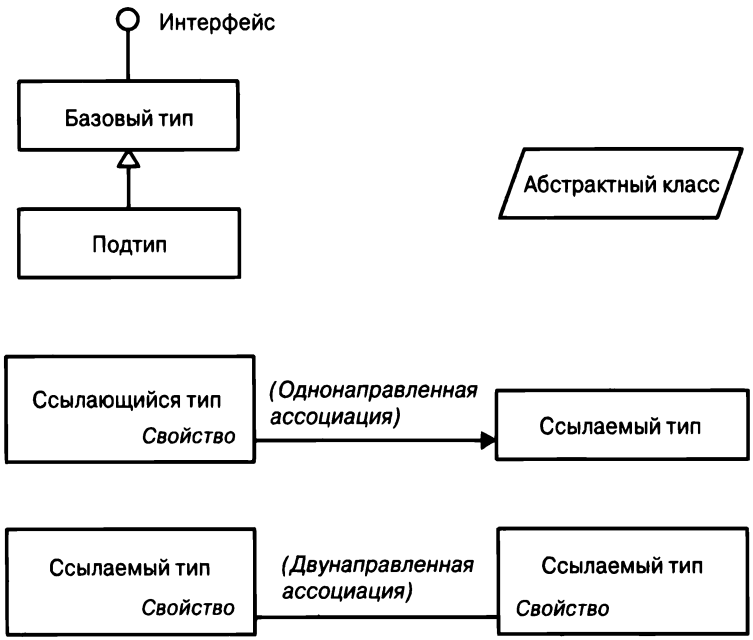



***Рис. 0.1.*** *Пример диаграммы*

В книге также приняты следующие типографские соглашения.

*Курсив*

Применяется для новых терминов.

`Моноширинный`

Применяется для кода C#, ключевых слов и идентификаторов, а также вывода из программ.

**`Моноширинный полужирный`**

Применяется для выделения части кода.

*`Моноширинный курсив`*

Применяется для выделения текста, который должен быть заменен значениями, предоставленными пользователем.

Здесь приводится совет, указание или общее замечание.

Здесь приводится предупреждение или предостережение.

# Использование примеров кода

Настоящая книга призвана помочь вам выполнять свою работу. Обычно, если в книге предлагается пример кода, то вы можете применять его в собственных программах и документации. Вы не обязаны обращаться к нам за разрешением, если только не используете значительную долю кода. Скажем, написание программы, в которой задействовано несколько фрагментов кода из этой книги, разрешения не требует. Для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение обязательно. Ответ на вопрос путем цитирования данной книги и ссылки на пример кода разрешения не требует. Для встраивания значительного объема примеров кода, рассмотренных в этой книге, в документацию по вашему продукту разрешение обязательно.

Мы высоко ценим указание авторства, хотя и не требуем этого. Установление авторства обычно включает название книги, фамилию и имя автора, издательство и номер ISBN. Например: "C# 7.0 in a Nutshell by Joseph Albahari and Ben Albahari (O'Reilly). Copyright 2018, Joseph Albahari and Ben Albahari, 978-1-491-98765-0".

Если вам кажется, что способ использования вами примеров кода выходит за законные рамки или упомянутые выше разрешения, тогда свяжитесь с нами по следующему адресу электронной почты: `permissions@oreilly.com`.

# Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Наш почтовый адрес:

195027, Санкт-Петербург, Магнитогорская ул., д. 30, п/я 116

# 1

# Введение в C# и .NET Framework

C# является универсальным, безопасным в отношении типов, объектно-ориентированным языком программирования. Цель C# заключается в обеспечении продуктивности работы программистов. Для этого в языке соблюдается баланс между простотой, выразительностью и производительностью. С самой первой версии главным архитектором языка C# был Андерс Хейлсберг (создатель Turbo Pascal и архитектор Delphi). Язык C# нейтрален в отношении платформ и работает с различными компиляторами, специфическими для платформ, наиболее заметной из которых считается Microsoft .NET Framework для Windows.

## Объектная ориентация

В C# предлагается расширенная реализация парадигмы объектной ориентации, которая включает *инкапсуляцию*, *наследование* и *полиморфизм*. Инкапсуляция означает создание вокруг объекта границы, предназначенной для отделения внешнего (открытого) поведения от внутренних (закрытых) деталей реализации. Ниже перечислены отличительные особенности языка C# с объектно-ориентированной точки зрения.

- **Унифицированная система типов.** Фундаментальным строительным блоком в C# является инкапсулированная единица данных и функций, которая называется *типом*. Язык C# имеет *унифицированную систему типов*, где все типы в конечном итоге разделяют общий базовый тип. Это значит, что все типы, независимо от того, представляют они бизнес-объекты или примитивные сущности вроде чисел, совместно используют одну и ту же базовую функциональность. Например, экземпляр любого типа может быть преобразован в строку вызовом его метода `ToString`.
- **Классы и интерфейсы.** В рамках традиционной объектно-ориентированной парадигмы единственной разновидностью типа является класс. В C# присутствуют типы других видов, один из которых — *интерфейс*. Интерфейс похож на класс за исключением того, что он только описывает члены. Реализация для этих членов поступает из типов, которые *реализуют* данный интерфейс. Интерфейсы особенно полезны в сценариях, когда требуется множественное наследование (в отли-

чие от таких языков, как C++ и Eiffel, множественное наследование классов в C# не поддерживается).

- Свойства, методы и события. В чистой объектно-ориентированной парадигме все функции представляют собой *методы* (это случай Smalltalk). В C# методы являются только одной разновидностью функций-членов, куда также относятся *свойства* и *события* (помимо прочих). Свойства — это функции-члены, которые инкапсулируют фрагмент состояния объекта, такой как цвет кнопки или текст метки. События — это функции-члены, упрощающие выполнение действий при изменении состояния объекта.

Хотя C# — главным образом объектно-ориентированный язык, он также заимствует кое-что из парадигмы *функционального программирования*. Конкретные заимствования перечислены ниже.

- **Функции могут трактоваться как значения.** За счет применения *делегатов* язык C# позволяет передавать функции как значения в другие функций и получать их из других функций.
- **Для чистоты в C# поддерживаются паттерны.** Основная черта функционального программирования заключается в том, чтобы избегать использования переменных, значения которых изменяются, отдавая предпочтение декларативным паттернам. В C# имеются ключевые средства, содействующие таким паттернам, в том числе возможность написания на лету неименованных функций, которые "захватывают" переменные (*лямбда-выражения*), и возможность спискового или реактивного программирования через *выражения запросов*. В C# 6.0 также легко определять поля и свойства только для чтения с целью написания *неизменяемых* (допускающих только чтение) типов.

# Безопасность в отношении типов

Прежде всего, C# является языком, *безопасным к типам*. Это означает, что экземпляры типов могут взаимодействовать только через определяемые ими протоколы, обеспечивая тем самым внутреннюю согласованность каждого типа. Например, C# не позволяет взаимодействовать со *строковым* типом так, как если бы он был *целочисленным* типом.

Точнее говоря, в C# поддерживается *статическая типизация*, при которой язык обеспечивает безопасность в отношении типов *на этапе компиляции*. Это делается в дополнение к безопасности в отношении типов, навязываемой *во время выполнения*.

Статическая типизация ликвидирует обширную категорию ошибок еще до запуска программы. Она перекладывает бремя проверки того, что все типы в программе корректно подходят друг другу, с модульных тестов времени выполнения на компилятор. В результате крупные программы становятся намного более легкими в управлении, более предсказуемыми и более надежными. Кроме того, статическая типизация позволяет таким инструментам, как IntelliSense в Visual Studio, оказывать помощь в написании программы: поскольку тип заданной переменной известен, то известны и методы, которые можно вызывать с этой переменной.

Язык C# также позволяет частям кода быть динамически типизированными через ключевое слово `dynamic`. Тем не менее, C# остается преимущественно статически типизированным языком.

Язык C# также называют *строго типизированным*, потому что его правила, касающиеся типов (применяемые как статически, так и во время выполнения), являются очень строгими. Например, невозможно вызвать функцию, которая предназначена для приема целого числа, с числом с плавающей точкой, не выполнив предварительно *явное* преобразование числа с плавающей точкой в целое. Это помогает предотвращать ошибки.

Строгая типизация также играет важную роль в обеспечении возможности запуска кода C# в *песочнице* – среде, где каждый аспект безопасности контролируется хостом. Важной особенностью песочницы является то, что вы не сможете своевольно разрушить состояние объекта за счет обхода связанных с ним правил типов.

## Управление памятью

В плане реализации автоматического управления памятью C# полагается на исполняющую среду. Общеязыковая исполняющая среда (Common Language Runtime – CLR) имеет сборщик мусора, выполняющийся как часть вашей программы; он восстанавливает память, занятую объектами, на которые больше нет ссылок. Это снимает с программистов обязанность явно освобождать память для объекта, устраняя проблему некорректных указателей, которая встречается в языках вроде C++.

Язык C# не исключает указатели: он просто делает их необязательными при решении большинства задач программирования. Для "горячих" точек, критичных к производительности, и возможности взаимодействия указатели и явное выделение памяти разрешены в блоках, которые явно помечены как `unsafe` (т.е. небезопасные).

## Поддержка платформ

Исторически сложилось так, что язык C# применялся в основном для написания кода, подлежащего выполнению на платформах Windows. Однако недавно Microsoft и ряд компаний инвестировали в другие платформы, включая Linux, macOS, iOS и Android. Инфраструктура Xamarin делает возможной межплатформенную разработку мобильных приложений на языке C#, а переносимые библиотеки классов (Portable Class Libraries) становятся все более широко распространенными. Межплатформенная легковесная инфраструктура веб-хостинга ASP.NET Core от Microsoft способна функционировать под управлением либо .NET Framework, либо .NET Core (межплатформенной исполняющей средой с открытым кодом).

## C# и CLR

Язык C# зависит от исполняющей среды, оснащенной многочисленными функциональными средствами, такими как автоматическое управление памятью и обработка исключений. Центральной частью Microsoft .NET Framework является *общеязыковая исполняющая среда* (CLR), которая предоставляет такие средства времени выполнения. (Инфраструктуры .NET Core и Xamarin предлагают похожие исполняющие среды.) Среда CLR нейтральна к языку, позволяя разработчикам строить приложения на многих языках (скажем, C#, F#, Visual Basic .NET и Managed C++).

C# – один из нескольких *управляемых языков*, которые компилируются в управляемый код. Управляемый код представлен на *промежуточном языке* (Intermediate Language – IL). Среда CLR преобразует код IL в собственный код машины, такой как X86 или X64, обычно прямо перед выполнением. Такое преобразование называется оперативной компиляцией (just-in-time – JIT). Доступна также ранняя (ahead-of-time)

компиляция для улучшения показателей времени начального запуска в случае крупных сборок или устройств с ограниченными ресурсами (а также для удовлетворения правил хранилища приложений iOS, когда разработка ведется с помощью Xamarin).

Контейнер для управляемого кода называется *сборкой* или *переносимым исполняемым модулем*. Сборка может быть исполняемым файлом (`.exe`) или библиотекой (`.dll`) и содержит не только код IL, но также информацию о типах (*метаданные*). Наличие метаданных дает возможность сборкам ссылаться на типы в других сборках, не нуждаясь в дополнительных файлах.

Вы можете исследовать и дизассемблировать содержимое сборки IL посредством инструмента `ildasm` от Microsoft. А такие инструменты, как ILSpy, dotPeek (JetBrains) или Reflector (Red Gate), позволяют пойти еще дальше и декомпилировать код IL в C#. Поскольку по сравнению с собственным машинным кодом код IL является более высокоуровневым, декомпилятор способен проделать довольно хорошую работу по воссозданию исходного кода C#.

Программа может запрашивать собственные метаданные (*рефлексия*) и даже генерировать новый код IL во время выполнения (`Reflection.Emit`).

# CLR и .NET Framework

Платформа .NET Framework образована из CLR и обширного набора библиотек. Библиотеки состоят из библиотек ядра (описанных в этой книге) и прикладных библиотек, которые зависят от библиотек ядра. На рис. 1.1 представлен визуальный обзор упомянутых библиотек (он также служит вспомогательным средством навигации по книге).

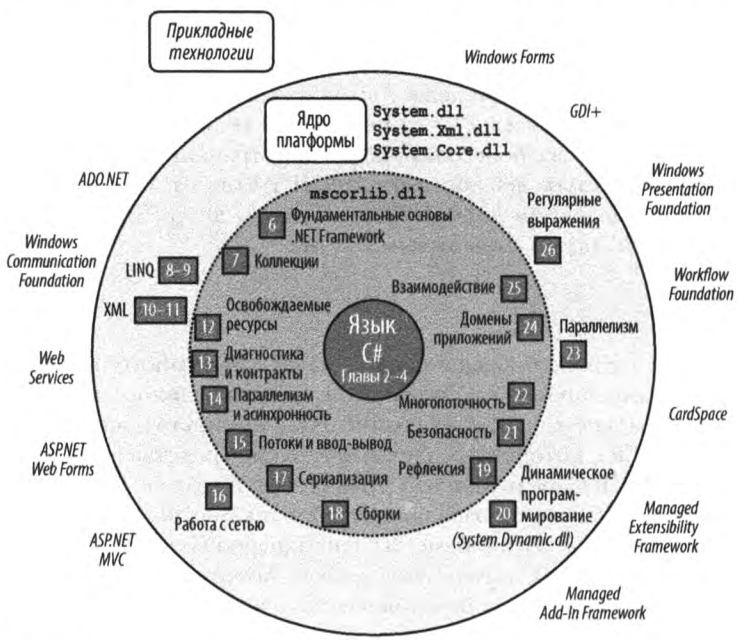



***Рис. 1.1.*** *Темы, рассмотренные в книге, и главы, в которых их можно найти. Темы, не раскрытые в книге, показаны за пределами большого круга*

Библиотеки ядра иногда все вместе называются *библиотекой базовых классов* (Base Class Library – BCL). Полная инфраструктура называется *библиотекой классов инфраструктуры* (Framework Class Library – FCL).

# Другие инфраструктуры

Инфраструктура Microsoft .NET Framework является наиболее обширной и зрелой, но функционирует только под управлением Microsoft Windows (на рабочей станции или сервере). С годами появились другие инфраструктуры, предназначенные для поддержки других платформ. В настоящее время помимо .NET Framework есть три основных игрока, которые все принадлежат Microsoft.

- **Универсальная платформа Windows (Universal Windows Platform — UWP).** Позволяет писать приложения для Windows 10 Store и нацеливать их на устройства Windows 10 (мобильные, XBox, Surface Hub, HoloLens). В целях уменьшения угрозы со стороны вредоносного программного обеспечения приложение выполняется в песочнице с запретом таких операций, как чтение или запись в произвольные файлы.
- **.NET Core с ASP.NET Core.** Инфраструктура с открытым кодом (первоначально основанная на усеченной версии .NET Framework), предназначенная для написания легко развертываемых Интернет-приложений и микросервисов, которые функционируют в среде Windows, macOS и Linux. В отличие от .NET Framework инфраструктура .NET Core может упаковываться с веб-приложением и развертываться методом xcopy (автономное развертывание).
- **Xamarin.** Для написания мобильных приложений, которые нацелены на iOS, Android и Windows Mobile. Компания Xamarin была приобретена Microsoft в 2016 году.

В табл. 1.1 сравнивается текущая поддержка платформ всеми основными инфраструктурами.

**Таблица 1.1. Поддержка платформ популярными инфраструктурами**

| Целевая операционная система | .NET Framework | UWP | .NET Core | Xamarin |
|---|---|---|---|---|
| Windows 7/8 | Да | | Да | |
| Windows 10 на рабочей станции или сервере | Да | Да | Да | |
| Устройства Windows 10 | | Да | | Да |
| Linux | | | Да | |
| macOS | | | Да | |
| iOS (iPhone) | | | | Да |
| Android | | | | Да |

Четыре главные инфраструктуры отличаются поддерживаемыми платформами, библиотеками, на которых они построены, и предполагаемым применением. Тем не менее, справедливо заявить, что с выпуском .NET Core 2.0 все они предоставляют доступ к похожей основной инфраструктуре (BCL), на которой сосредоточено главное внимание в данной

книге. Можно даже прямо воспользоваться этой общностью в своих интересах, написав библиотеки классов, которые работают во всех четырех инфраструктурах (см. раздел ".NET Standard 2.0" в главе 5).

В табл. 1.1 не был отражен один нюанс: UWP внутренне задействует .NET Core, так что формально .NET Core работает на устройствах Windows 10 (хотя и не в целях предоставления инфраструктуры для ASP.NET Core). Вероятно, в будущем мы увидим больше возможностей для применения .NET Core 2.

## Унаследованные и нишевые инфраструктуры

Перечисленные ниже инфраструктуры по-прежнему доступны для поддержки старых платформ:

- Windows Runtime для Windows 8/8.1 (теперь UWP);
- Windows Phone 7/8 (теперь UWP);
- Microsoft XNA для разработки игр (теперь UWP);
- Silverlight (после подъема интереса к HTML5 и JavaScript больше активно не развивается);
- .NET Core 1.x (предшественница .NET Core 2.0 со значительно усеченной функциональностью).

Есть также пара нишевых инфраструктур, стоящих упоминания.

- Инфраструктура .NET Micro Framework предназначена для выполнения кода .NET на встроенных устройствах с крайне ограниченными ресурсами (до 1 Мбайт).
- Инфраструктура с открытым кодом Mono, на которой основана Xamarin, также имеет библиотеки, позволяющие разрабатывать межплатформенные настольные приложения для Linux, macOS и Windows. Не все функциональные средства поддерживаются или полностью работают.

Возможно также выполнение управляемого кода внутри SQL Server. Благодаря интеграции CLR с SQL Server вы можете писать специальные функции, хранимые процедуры и агрегации на языке C# и затем вызывать их в коде SQL. Такой прием работает в сочетании со стандартной инфраструктурой .NET Framework, но со специальной "размещенной" средой CLR, которая обеспечивает песочницу для защиты целостности процесса SQL Server.

## Windows Runtime

Язык C# также взаимодействует с технологией *Windows Runtime* (WinRT). С WinRT связаны две вещи:

- нейтральный к языку объектно-ориентированный интерфейс выполнения, поддерживаемый в Windows 8 и последующих версиях;
- набор библиотек, встроенных в Windows 8 и последующие версии, которые поддерживают предыдущий интерфейс.

Привнося некоторую путаницу, термин WinRT исторически использовался для обозначения еще двух вещей:

- предшественницы UWP, т.е. платформы разработки для написания приложений Windows 8/8.1 Store, иногда называемой Metro или Modern;
- исчезнувшей операционной системы для планшетов на основе RISC-процессоров (Windows RT), которую Microsoft выпустила в 2011 году.

Под *интерфейсом выполнения* понимается протокол для вызывающего кода, который (потенциально) написан на другом языке. Исторически сложилось так, что операционная система Microsoft Windows предоставляет примитивный интерфейс выполнения в форме низкоуровневых вызовов функций в стиле C, содержащихся в Win32 API.

В среде WinRT ситуация гораздо лучше. Отчасти WinRT представляет собой расширенную версию COM (Component Object Model – модель компонентных объектов), которая поддерживает .NET, C++ и JavaScript. В отличие от Win32 она является объектно-ориентированной и располагает относительно развитой системой типов. Это означает, что ссылка на библиотеку WinRT из C# воспринимается во многом подобно ссылке на библиотеку .NET – вы можете даже не осознавать, что применяете WinRT.

Библиотеки WinRT в Windows 10 образуют существенную часть платформы UWP (UWP полагается на библиотеки WinRT и .NET Core). Если вы нацеливаетесь на стандартную платформу .NET Framework, тогда ссылка на библиотеки Windows 10 WinRT необязательна, но может быть полезной, когда необходим доступ к функциональным средствам, специфическим для Windows 10, которые по-другому не раскрыты в .NET Framework.

Библиотеки WinRT в Windows 10 поддерживают пользовательский интерфейс UWP для написания многонаправленных (т.е. одновременно воздействующих на человека через несколько каналов восприятия) сенсорных приложений. Они также поддерживают возможности, специфичные для мобильных устройств, такие как датчики, текстограммы (короткие текстовые сообщения, отправляемые по SMS) и т.д. (новая функциональность Window 8, 8.1 и 10 открывается через WinRT, а не Win32). Библиотеки WinRT также предоставляют функции файлового ввода-вывода, специально приспособленные для эффективной работы внутри песочницы UWP.

Отличие от обычного COM состоит в том, что интерфейс WinRT *проецирует* свои библиотеки на множество языков (C#, VB, C++ и JavaScript), поэтому каждый язык видит типы WinRT почти так, как если бы они были реализованы специально для него. Например, WinRT будет адаптировать правила применения заглавных букв, чтобы соответствовать стандартам целевого языка, и даже переназначать некоторые функции и интерфейсы. Сборки WinRT также поставляются с расширенными *метаданными* в файлах `.winmd`, которые имеют тот же формат, что и файлы сборок .NET, делая возможным их прозрачное потребление без специального ритуала. Именно потому вы можете не знать, что используете типы WinRT, а не типы .NET, если оставить в стороне несходства пространств имен. Еще один ключевой момент связан с тем, что типы WinRT подвержены ограничениям в стиле COM; скажем, они предлагают ограниченную поддержку наследования и обобщений.

В языке C# вы можете не только потреблять библиотеки WinRT, но также писать собственные библиотеки (и обращаться к ним из приложения JavaScript).

# Краткая история языка C#

Далее приводится обратная хронология появления новых средств в каждой версии C#, которая будет полезна читателям, знакомым с более старыми версиями языка.

## Нововведения версии C# 7.0

(Версия C# 7.0 поставляется вместе с *Visual Studio 2017*.)

### Улучшения числовых литералов

Для улучшения читабельности числовые литералы в C# 7 могут включать символы подчеркивания, которые называются *разделителями групп разрядов* и компилятором игнорируются:

```
int million = 1_000_000;
```

С помощью префикса 0b могут указываться *двоичные литералы*:

```
var b = 0b1010_1011_1100_1101_1110_1111;
```

### Переменные out и отбрасывание

В версии C# 7 облегчается вызов методов, которые содержат параметры out. Прежде всего, теперь вы можете объявлять *переменные out* на лету:

```
bool successful = int.TryParse ("123", out int result);
Console.WriteLine (result);
```

А при вызове метода с множеством параметров out с помощью символа подчеркивания вы можете отбрасывать те из них, в которых не заинтересованы:

```
SomeBigMethod (out _, out _, out _, out int x, out _, out _, out _);
Console.WriteLine (x);
```

### Шаблоны

Посредством операции is вы также можете вводить переменные на лету. Они называются *шаблонными переменными* (см. раздел "Операция is и шаблонные переменные (C# 7)" в главе 3):

```
void Foo (object x)
{
  if (x is string s)
    Console.WriteLine (s.Length);
}
```

Оператор switch также поддерживает шаблоны, поэтому вы можете переключаться на основе *типа*, а также на основе констант (см. раздел "Оператор switch с шаблонами (C# 7) в главе 2). Вы можете указывать условия в конструкции when и также переключаться по значению null:

```
switch (x)
{
  case int i:
    Console.WriteLine ("Это целочисленное значение!");
    break;
  case string s:
    Console.WriteLine (s.Length);  // Можно использовать переменную s
    break;
```

```
  case bool b when b == true:  // Соответствует, только когда b равно true
    Console.WriteLine ("True");
    break;
  case null:
    Console.WriteLine ("Ничего не делать");
    break;
}
```

### Локальные методы

*Локальный метод* — это метод, объявленный внутри другой функции (см. раздел "Локальные методы (C# 7)" в главе 3):

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  Console.WriteLine (Cube (4));
  Console.WriteLine (Cube (5));
  int Cube (int value) => value * value * value;
}
```

Локальные методы видны только вмещающей функции и могут захватывать локальные переменные тем же способом, что и лямбда-выражения.

### Больше членов, сжатых до выражений

В версии C# 6 был введен синтаксис сжатия до выражений (=>) для методов, свойств только для чтения, операций и индексаторов. В версии C# 7 он расширяется на конструкторы, свойства для чтения/записи и финализаторы:

```
public class Person
{
  string name;

  public Person (string name) => Name = name;

  public string Name
  {
    get => name;
    set => name = value ?? "";
  }
  ~Person () => Console.WriteLine ("финализация");
}
```

### Деконструкторы

В версии C# 7 появился шаблон *деконструирования*. В то время как конструктор обычно принимает набор значений (в качестве параметров) и присваивает их полям, *деконструктор* делает противоположное, присваивая поля обратно набору переменных. Деконструктор для класса `Person` из предыдущего примера можно было бы написать следующим образом (обработка исключений опущена):

```
public void Deconstruct (out string firstName, out string lastName)
{
  int spacePos = name.IndexOf (' ');
  firstName = name.Substring (0, spacePos);
  lastName = name.Substring (spacePos + 1);
}
```

Деконструкторы вызываются с помощью специального синтаксиса:

```
var joe = new Person ("Joe Bloggs");
var (first, last) = joe;      // Деконструирование
Console.WriteLine (first);    // Joe
Console.WriteLine (last);     // Bloggs
```

## Кортежи

Пожалуй, самым заметным усовершенствованием, внесенным в версию C# 7, является явная поддержка кортежей (см. раздел "Кортежи (C# 7)" в главе 4). Кортежи предоставляют простой способ хранения набора связанных значений:

```
var bob = ("Bob", 23);
Console.WriteLine (bob.Item1);  // Bob
Console.WriteLine (bob.Item2);  // 23
```

Кортежи в C# представляют собой "синтаксический сахар" для использования обобщенных структур `System.ValueTuple<...>`. Но благодаря магии компилятора элементы кортежа могут быть именованными:

```
var tuple = (Name:"Bob", Age:23);
Console.WriteLine (tuple.Name);  // Bob
Console.WriteLine (tuple.Age);   // 23
```

С появлением кортежей функции могут возвращать множество значений без обращения к параметрам `out`:

```
static (int row, int column) GetFilePosition() => (3, 10);

static void Main()
{
  var pos = GetFilePosition();
  Console.WriteLine (pos.row);     // 3
  Console.WriteLine (pos.column);  // 10
}
```

Кортежи неявно поддерживают шаблон деконструирования, поэтому их легко *деконструировать* в индивидуальные переменные. Предыдущий метод `Main` можно переписать так, чтобы кортеж, возвращаемый методом `GetFilePosition`, взамен присваивался двум локальным переменным, `row` и `column`:

```
static void Main()
{
  (int row, int column) = GetFilePosition(); // Создает 2 локальные переменные
  Console.WriteLine (row);                   // 3
  Console.WriteLine (column);                // 10
}
```

## Выражения `throw`

До выхода версии C# 7 конструкция `throw` всегда была оператором. Теперь она может также появляться как выражение в функциях, сжатых до выражения:

```
public string Foo() => throw new NotImplementedException();
```

Выражение `throw` может также находиться внутри тернарной условной операции:

```
string Capitalize (string value) =>
  value == null ? throw new ArgumentException ("value") :
  value == "" ? "" :
  char.ToUpper (value[0]) + value.Substring (1);
```

### Другие улучшения

Версия C# 7 включает также пару средств для специализированных сценариев микро-оптимизации (см. разделы "Ссылочные локальные переменные (C# 7)" и "Возвращаемые ссылочные значения (C# 7)" в главе 2) и возможность объявлять асинхронные методы с возвращаемыми типами, которые отличаются от `Task/Task<T>`.

## Нововведения версии C# 6.0

Особенностью версии C# 6.0, поставляемой в составе *Visual Studio 2015*, стал компилятор нового поколения, который был полностью написан на языке C#. Известный как проект "Roslyn", новый компилятор делает видимым весь конвейер компиляции через библиотеки, позволяя проводить кодовый анализ для произвольного исходного кода (глава 27). Сам компилятор представляет собой проект с открытым кодом, и его исходный код доступен по адресу `github.com/dotnet/roslyn`.

Кроме того, в версии C# 6.0 появилось несколько небольших, но важных усовершенствований, направленных главным образом на сокращение беспорядка в коде.

`null`-*условная операция* (или элвис-*операция*), рассматриваемая в главе 2, устраняет необходимость в явной проверке на равенство `null` перед вызовом метода или доступом к члену типа. В следующем примере вместо генерации исключения `NullReferenceException` переменная `result` получает значение `null`:

```
System.Text.StringBuilder sb = null;
string result = sb?.ToString();  // Переменная result равна null
```

*Функции, сжатые до выражений* (expression-bodied function), которые обсуждаются в главе 3, дают возможность записывать методы, свойства, операции и индексаторы, содержащие единственное выражение, более компактно в стиле лямбда-выражений:

```
public int TimesTwo (int x) => x * 2;
public string SomeProperty => "Property value";
```

*Инициализаторы свойств* (глава 3) позволяют присваивать начальные значения автоматическим свойствам:

```
public DateTime Created { get; set; } = DateTime.Now;
```

Инициализируемые свойства также могут быть допускающими только чтение:

```
public DateTime Created { get; } = DateTime.Now;
```

Свойства, предназначенные только для чтения, можно также устанавливать в конструкторе, что облегчает создание неизменяемых (допускающих только чтение) типов.

*Инициализаторы индексов* (глава 4) делают возможной инициализацию за один шаг для любого типа, который открывает доступ к индексатору:

```
new Dictionary<int,string>()
{
  [3] = "three",
  [10] = "ten"
}
```

*Интерполяция строк* (глава 2) предлагает лаконичную альтернативу вызову метода `string.Format`:

```
string s = $"It is {DateTime.Now.DayOfWeek} today";
```

*Фильтры исключений* (глава 4) позволяют применять условия к блокам `catch`:

```
try
{
  new WebClient().DownloadString("http://asef");
}
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
  ...
}
```

Директива `using static` (глава 2) позволяет импортировать все статические члены типа, так что такими членами можно пользоваться без уточнения имени типа:

```
using static System.Console;
...
WriteLine ("Hello, world");   // WriteLine вместо Console.WriteLine
```

Операция `nameof` (глава 3) возвращает имя переменной, типа или другого символа в виде строки. Это препятствует нарушению работы кода, когда переименовывается какой-то символ в Visual Studio:

```
int capacity = 123;
string x = nameof (capacity); // x имеет значение "capacity"
string y = nameof (Uri.Host); // y имеет значение "Host"
```

Наконец, теперь разрешено применять `await` внутри блоков `catch` и `finally`.

# Нововведения версии C# 5.0

Крупным нововведением версии C# 5.0 была поддержка *асинхронных функций* с помощью двух новых ключевых слов, `async` и `await`. Асинхронные функции делают возможными *асинхронные продолжения*, которые облегчают написание быстрореагирующих и безопасных к потокам обогащенных клиентских приложений. Они также упрощают написание эффективных приложений с высоким уровнем параллелизма и интенсивным вводом-выводом, которые не связывают потоковый ресурс при выполнении операций.

Асинхронные функции подробно рассматриваются в главе 14.

# Нововведения версии C# 4.0

Ниже перечислены средства, которые появились в C# 4.0:

- динамическое связывание;
- необязательные параметры и именованные аргументы;
- вариантность типов с обобщенными интерфейсами и делегатами;
- улучшение взаимодействия с COM.

*Динамическое связывание* (главы 4 и 20) откладывает *связывание* – процесс распознавания типов и членов – с этапа компиляции до времени выполнения и полезно в сценариях, которые иначе требовали бы сложного кода рефлексии. Динамическое связывание также удобно при взаимодействии с динамическими языками и компонентами COM.

*Необязательные параметры* (глава 2) позволяют функциям указывать стандартные значения параметров, так что в вызывающем коде аргументы можно опускать. *Именованные аргументы* дают возможность вызывающему коду идентифицировать аргумент по имени, а не по позиции.

Правила *вариантности типов* в C# 4.0 были ослаблены (главы 3 и 4), так что параметры типа в обобщенных интерфейсах и обобщенных делегатах могут помечаться как *ковариантные* или *контравариантные*, делая возможными более естественные преобразования типов.

*Взаимодействие с COM* (глава 25) в C# 4.0 было улучшено в трех отношениях. Во-первых, аргументы могут передаваться по ссылке без ключевого свойства `ref` (что особенно удобно в сочетании с необязательными параметрами). Во-вторых, сборки, которые содержат типы взаимодействия с COM, можно *связывать*, а не *ссылаться* на них. Связанные типы взаимодействия поддерживают эквивалентность типов, устранив необходимость в наличии *основных сборок взаимодействия* (Primary Interop Assembly) и положив конец мучениям с ведением версий и развертыванием. В-третьих, функции, которые возвращают COM-типы `variant` из связанных типов взаимодействия, отображаются на тип `dynamic`, а не `object`, ликвидируя потребность в приведении.

# Нововведения версии C# 3.0

Средства, добавленные в версии C# 3.0, по большей части были сосредоточены на возможностях *языка интегрированных запросов* (Language Integrated Query – LINQ). Язык LINQ позволяет записывать запросы прямо внутри программы C# и *статически* проверять их корректность, при этом допуская запросы как к локальным коллекциям (вроде списков или документов XML), так и к удаленным источникам данных (подобным базам данных). Средства, которые были добавлены в версию C# 3.0 для поддержки LINQ, включают в себя неявно типизированные локальные переменные, анонимные типы, инициализаторы объектов, лямбда-выражения, расширяющие методы, выражения запросов и деревья выражений.

*Неявно типизированные локальные переменные* (ключевое слово `var`; глава 2) позволяют опускать тип переменной в операторе объявления, разрешая компилятору выводить его самостоятельно. Это снижает перегруженность, а также делает возможными *анонимные типы* (глава 4), которые представляют собой простые классы, создаваемые на лету и обычно применяемые в финальном выводе запросов LINQ. Массивы также могут быть неявно типизированными (см. главу 2).

*Инициализаторы объектов* (глава 3) упрощают конструирование объектов, позволяя устанавливать свойства прямо в вызове конструктора. Инициализаторы объектов работают как с именованными, так и с анонимными типами.

*Лямбда-выражения* (глава 4) – это миниатюрные функции, создаваемые компилятором на лету, которые особенно удобны в "текучем" синтаксисе запросов LINQ (глава 8).

*Расширяющие методы* (глава 4) расширяют существующий тип новыми методами (не изменяя определение типа) и делают статические методы похожими на методы экземпляра. Операции запросов LINQ реализованы как расширяющие методы.

*Выражения запросов* (глава 8) предоставляют высокоуровневый синтаксис для написания запросов LINQ, которые могут быть существенно проще при работе с множеством последовательностей или переменных диапазонов.

*Деревья выражений* (глава 8) – это миниатюрные кодовые модели документных объектов (Document Object Model – DOM), которые описывают лямбда-выражения, присвоенные переменным специального типа `Expression<TDelegate>`. Деревья выражений позволяют запросам LINQ выполняться удаленно (например, на сервере базы данных), поскольку их можно анализировать и транслировать во время выполнения (скажем, в оператор SQL).

В C# 3.0 также были добавлены автоматические свойства и частичные методы.

*Автоматические свойства* (глава 3) сокращают работу по написанию свойств, которые просто читают и устанавливают закрытое поддерживающее поле, заставляя компилятор делать все автоматически. *Частичные методы* (глава 3) позволяют автоматически сгенерированному частичному классу предоставлять настраиваемые привязки для ручного написания кода, который "исчезает" в случае, если не используется.

# Нововведения версии C# 2.0

Крупными нововведениями в версии C# 2 были обобщения (глава 3), типы, допускающие значение `null` (глава 4), итераторы (глава 4) и анонимные методы (предшественники лямбда-выражений). Перечисленные средства подготовили почву для введения LINQ в версии C# 3.

В C# 2 также была добавлена поддержка частичных классов, статических классов и множества мелких разносторонних средств, таких как квалификатор псевдонима пространства имен, дружественные сборки и буферы фиксированных размеров.

Введение обобщений потребовало новой среды CLR (CLR 2.0), поскольку обобщения поддерживают полную точность типов во время выполнения.

# 2

# Основы языка C#

В настоящей главе вы ознакомитесь с основами языка C#.

Все программы и фрагменты кода в этой и последующих двух главах доступны как интерактивные примеры для LINQPad. Проработка примеров в сочетании с чтением данной книги ускоряет процесс изучения, т.к. вы можете редактировать код примеров и немедленно видеть результаты без необходимости в настройке проектов и решений в Visual Studio.

Для загрузки примеров перейдите на вкладку Samples (Примеры) в окне LINQPad, щелкните на ссылке Download/import more samples (Загрузить/импортировать дополнительные примеры) и в открывшемся окне щелкните на ссылке Download C# 7.0 in a Nutshell samples (Загрузить примеры для книги *C# 7.0 in a Nutshell*). Утилита LINQPad бесплатна и доступна для загрузки на веб-сайте `http://www.linqpad.net`.

## Первая программа на C#

Ниже показана программа, которая перемножает 12 и 30, после чего выводит на экран результат 360. Двойная косая черта (`//`) указывает на то, что остаток строки является *комментарием*.

```
using System;                   // Импортирование пространства имен

class Test                      // Объявление класса
{
  static void Main()            //   Объявление метода
  {
    int x = 12 * 30;            //     Оператор 1
    Console.WriteLine (x);      //     Оператор 2
  }                             //   Конец метода
}                               // Конец класса
```

В основе программы лежат два *оператора*:

```
int x = 12 * 30;
Console.WriteLine (x);
```

Операторы в C# выполняются последовательно и завершаются точкой с запятой (или, как вы вскоре увидите, образуют *блок кода*). Первый оператор вычисляет *выражение* `12 * 30` и сохраняет результат в *локальной переменной* по имени `x`, которая имеет

целочисленный тип. Второй оператор вызывает *метод* `WriteLine` класса `Console` для вывода значения переменной `x` в текстовое окно на экране.

*Метод* выполняет действие в виде последовательности операторов, которая называется *блоком операторов* и представляет собой пару фигурных скобок, содержащих ноль или большее количество операторов. Мы определили единственный метод по имени `Main`:

```
static void Main()
{
  ...
}
```

Написание функций более высокого уровня, которые вызывают функции более низких уровней, упрощает программу. Мы можем провести *рефакторинг* программы, выделив многократно используемый метод, который умножает целое число на 12, как показано ниже:

```
using System;

class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30));   // 360
    Console.WriteLine (FeetToInches (100));  // 1200
  }
  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

За счет указания параметров метод может получать *входные* данные из вызывающего кода, а за счет указания *возвращаемого типа* — передавать *выходные* данные обратно в вызывающий код. Мы определили метод по имени `FeetToInches`, который имеет параметр для входного значения в футах и возвращаемый тип для выходного значения в дюймах:

```
static int FeetToInches (int feet) {...}
```

*Литералы* `30` и `100` — это *аргументы*, передаваемые методу `FeetToInches`. Метод `Main` в рассматриваемом примере содержит пустые круглые скобки, поскольку он не принимает параметров, и является `void`, т.к. не возвращает какого-либо значения вызывающему коду:

```
static void Main()
```

Метод по имени `Main` распознается компилятором C# как признак стандартной точки входа в поток выполнения. Метод `Main` может дополнительно возвращать целочисленное значение (вместо того, чтобы быть `void`) с целью его передачи исполняющей среде (причем ненулевое значение, как правило, указывает на ошибку). Кроме того, метод `Main` может дополнительно принимать в качестве параметра массив строк (который будет заполняться любыми аргументами, передаваемыми исполняемому файлу).

Например:

```
static int Main (string[] args) {...}
```

Массив (такой как `string[]`) представляет фиксированное количество элементов определенного типа. Массивы указываются путем помещения квадратных скобок после типа их элементов и описаны в разделе "Массивы" далее в главе.

Методы являются одним из нескольких видов функций в C#. Другим видом функции, который мы применяли в примере программы, была *операция* `*`, выполняющая умножение. Существуют также *конструкторы*, *свойства*, *события*, *индексаторы* и *финализаторы*.

В приведенном выше примере два метода сгруппированы в класс. *Класс* объединяет функции-члены и данные-члены с целью формирования объектно-ориентированного строительного блока. Класс `Console` группирует члены, которые поддерживают функциональность ввода-вывода в командной строке, такие как метод `WriteLine`. Наш класс `Test` объединяет два метода — `Main` и `FeetToInches`. Класс — это разновидность *типа*, как обсуждается в разделе "Основы типов" далее в главе.

На самом внешнем уровне программы типы организованы в *пространства имен*. Директива `using` была использована для того, чтобы сделать пространство имен `System` доступным нашему приложению и работать с классом `Console`. Мы могли бы определить все свои классы внутри пространства имен `TestPrograms`, как показано ниже:

```
using System;

namespace TestPrograms
{
  class Test  {...}
  class Test2 {...}
}
```

Инфраструктура .NET Framework организована в виде вложенных пространств имен. Например, следующее пространство имен содержит типы для обработки текста:

```
using System.Text;
```

Директива `using` присутствует здесь ради удобства; на тип можно также ссылаться с помощью полностью заданного имени, которое представляет собой имя типа, предваренное его пространством имен, наподобие `System.Text.StringBuilder`.

## Компиляция

Компилятор C# транслирует исходный код, указываемый в виде набора файлов с расширением `.cs`, в *сборку* (assembly). Сборка — это единица упаковки и развертывания в .NET. Сборка может быть либо *приложением*, либо *библиотекой*. Нормальное консольное или Windows-приложение имеет метод `Main` и является файлом `.exe`. Библиотека представляет собой файл `.dll` и эквивалентна файлу `.exe` без точки входа. Она предназначена для вызова (*ссылки*) приложением или другими библиотеками. Инфраструктура .NET Framework — это набор библиотек.

Компилятор C# имеет имя *`csc.exe`*. Для выполнения компиляции можно либо пользоваться IDE-средой, такой как Visual Studio, либо запускать `csc` вручную в командной строке. (Компилятор также доступен в форме библиотеки; см. главу 27.)

Для компиляции вручную сначала необходимо сохранить программу в файл, такой как `MyFirstProgram.cs`, после чего перейти в окно командной строки и запустить компилятор `csc` (расположенный в `%Program Files(X86)%\msbuild\15.0\bin`) следующим образом:

```
csc MyFirstProgram.cs
```

В результате будет получено приложение по имени `MyFirstProgram.exe`.

Несколько необычно, но версии .NET Framework 4.6 и 4.7 все еще поставляются с компилятором C# 5. Чтобы получить компилятор командной строки для версии C# 7, понадобится установить Visual Studio 2017 или MSBuild 15.

Для построения библиотеки (*.dll*) воспользуйтесь такой командой:

```
csc /target:library MyFirstProgram.cs
```

Сборки подробно рассматриваются в главе 18.

# Синтаксис

На синтаксис C# оказал влияние синтаксис языков C и C++. В этом разделе мы опишем элементы синтаксиса C#, применяя в качестве примера следующую программу:

```
using System;

class Test
{
  static void Main()
  {
    int x = 12 * 30;
    Console.WriteLine (x);
  }
}
```

## Идентификаторы и ключевые слова

*Идентификаторы* – это имена, которые программисты выбирают для своих классов, методов, переменных и т.д. Ниже перечислены идентификаторы из примера программы в порядке их появления:

```
System  Test  Main  x  Console  WriteLine
```

Идентификатор должен быть целостным словом, которое по существу состоит из символов Unicode и начинается с буквы или символа подчеркивания. Идентификаторы C# чувствительны к регистру символов. По соглашению для параметров, локальных переменных и закрытых полей должен применяться "верблюжий" стиль (наподобие `myVariable`), а для всех остальных идентификаторов – стиль Pascal (вроде `MyMethod`).

*Ключевые слова* представляют собой имена, которые имеют для компилятора особый смысл. Ниже перечислены ключевые слова в нашем примере программы:

```
using  class  static  void  int
```

Большинство ключевых слов являются зарезервированными, а это означает, что их нельзя использовать в качестве идентификаторов. Вот полный список зарезервированных ключевых слов C#:

```
abstract    event       new         struct
as          explicit    null        switch
base        extern      object      this
bool        false       operator    throw
break       finally     out         true
byte        fixed       override    try
case        float       params      typeof
catch       for         private     uint
char        foreach     protected   ulong
checked     goto        public      unchecked
class       if          readonly    unsafe
const       implicit    ref         ushort
continue    in          return      using
decimal     int         sbyte       virtual
default     interface   sealed      void
delegate    internal    short       volatile
do          is          sizeof      while
double      lock        stackalloc
else        long        static
enum        namespace   string
```

### Устранение конфликтов

Если вы действительно хотите применять идентификатор с именем, которое конфликтует с ключевым словом, то к нему необходимо добавить префикс `@`. Например:

```
class class  {...}  // Не допускается
class @class {...}  // Разрешено
```

Символ `@` не является частью самого идентификатора. Таким образом, `@myVariable` – то же, что и `myVariable`.

Префикс `@` может быть полезен при потреблении библиотек, написанных на других языках .NET, в которых используются отличающиеся ключевые слова.

### Контекстные ключевые слова

Некоторые ключевые слова являются *контекстными*, а это значит, что их можно применять также в качестве идентификаторов – без символа `@`. Вот эти ключевые слова:

```
add          from        nameof      var
ascending    get         on          when
async        global      orderby     where
await        group       partial     yield
by           in          remove
descending   into        select
dynamic      join        set
equals       let         value
```

Неоднозначность с контекстными ключевыми словами не может возникать внутри контекста, в котором они используются.

## Литералы, знаки пунктуации и операции

*Литералы* — это элементарные порции данных, лексически встраиваемые в программу. В рассматриваемом примере программы присутствуют литералы 12 и 30.

*Знаки пунктуации* помогают размечать структуру программы. В рассматриваемом примере программы применяются следующие знаки пунктуации:

```
{  }  ;
```

Фигурные скобки группируют множество операторов в *блок операторов*.

Точка с запятой завершает оператор. (Тем не менее, блоки операторов не требуют в конце точки с запятой.) Операторы могут записываться в нескольких строках:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

*Операция* преобразует и объединяет выражения. Большинство операций в C# обозначаются с помощью некоторого символа, например, операция умножения выглядит как *. Ниже перечислены операции, задействованные в примере программы:

```
.  ()  *  =
```

Точка обозначает членство (или десятичную точку в числовых литералах). Круглые скобки используются при объявлении или вызове метода; пустые круглые скобки указываются, когда метод не принимает аргументов. (Позже в главе вы увидите, что круглые скобки имеют и другие предназначения.) Знак "равно" выполняет *присваивание*. (Двойной знак "равно", ==, производит сравнение эквивалентности.)

## Комментарии

В C# поддерживаются два разных стиля документирования исходного кода: *однострочные комментарии* и *многострочные комментарии*. Однострочный комментарий начинается с двойной косой черты и продолжается до конца строки. Например:

```
int x = 3;  // Комментарий относительно присваивания 3 переменной x
```

Многострочный комментарий начинается с символов /* и заканчивается символами */. Например:

```
int x = 3;  /* Это комментарий, который
               занимает две строки */
```

В комментарии могут быть встроены XML-дескрипторы документации, которые объясняются в разделе "XML-документация" главы 4.

# Основы типов

*Тип* определяет шаблон для значения. В рассматриваемом примере мы применяем два литерала типа int со значениями 12 и 30. Мы также объявляем *переменную* типа int по имени x:

```
static void Main()
{
  int x = 12 * 30;
  Console.WriteLine (x);
}
```

*Переменная* обозначает ячейку в памяти, которая с течением времени может содержать разные значения. Напротив, *константа* всегда представляет одно и то же значение (подробнее об этом — позже):

```
const int y = 360;
```

Все значения в C# являются *экземплярами* какого-то типа. Смысл значения и набор возможных значений, которые способна иметь переменная, определяются ее типом.

## Примеры предопределенных типов

Предопределенные типы — это типы, которые имеют специальную поддержку в компиляторе. Тип `int` является предопределенным типом для представления набора целых чисел, которые умещаются в 32 бита памяти, от $-2^{31}$ до $2^{31}-1$, и стандартным типом для числовых литералов в рамках указанного диапазона. С экземплярами типа `int` можно выполнять функции, например, арифметические:

```
int x = 12 * 30;
```

Еще один предопределенный тип в C# — `string`. Тип `string` представляет последовательность символов, такую как ".NET" или "http://oreilly.com". Со строками можно работать, вызывая для них функции следующим образом:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);  // HELLO WORLD

int x = 2018;
message = message + x.ToString();
Console.WriteLine (message);       // Hello world2018
```

Предопределенный тип `bool` поддерживает в точности два возможных значения: `true` и `false`. Тип `bool` обычно используется для условного разветвления потока выполнения с помощью оператора `if`. Например:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");  // Это не выводится

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");      // Это будет выведено
```

В языке C# предопределенные типы (также называемые встроенными типами) распознаются по ключевым словам C#. Пространство имен `System` в .NET Framework содержит много важных типов, которые не являются предопределенными в C# (скажем, `DateTime`).

## Примеры специальных типов

Точно так же, как из простых функций можно строить сложные функции, из примитивных типов допускается создавать сложные типы. В следующем примере мы определим специальный тип по имени `UnitConverter` — класс, который служит шаблоном для преобразования единиц:

```
using System;

public class UnitConverter
{
  int ratio;                                                    // Поле
  public UnitConverter (int unitRatio) { ratio = unitRatio; }  // Конструктор
  public int Convert   (int unit)      { return unit * ratio; } // Метод
}
class Test
{
  static void Main()
  {
    UnitConverter feetToInchesConverter = new UnitConverter (12);
    UnitConverter milesToFeetConverter  = new UnitConverter (5280);

    Console.WriteLine (feetToInchesConverter.Convert(30));     // 360
    Console.WriteLine (feetToInchesConverter.Convert(100));    // 1200
    Console.WriteLine (feetToInchesConverter.Convert(
                       milesToFeetConverter.Convert(1)));      // 63360
  }
}
```

### Члены типа

Тип содержит *данные-члены* и *функции-члены*. Данными-членами в типе `UnitConverter` является *поле* по имени `ratio`. Функции-члены в типе `UnitConverter` — это метод `Convert` и *конструктор* `UnitConverter`.

### Симметричность предопределенных и специальных типов

Привлекательный аспект языка C# заключается в том, что между предопределенными и специальными типами имеется лишь несколько отличий. Предопределенный тип `int` служит шаблоном для целых чисел. Он содержит данные — 32 бита — и предоставляет функции-члены, работающие с этими данными, такие как `ToString`. Аналогичным образом наш специальный тип `UnitConverter` действует в качестве шаблона для преобразований единиц. Он хранит данные — коэффициент (`ratio`) — и предоставляет функции-члены для работы с этими данными.

### Конструкторы и создание экземпляров

Данные создаются путем *создания экземпляров* типа. Создавать экземпляры предопределенных типов можно просто за счет применения литерала вроде `12` или `"Hello, world"`. Экземпляры специального типа создаются через операцию `new`. Мы объявляли и создавали экземпляр типа `UnitConverter` с помощью следующего оператора:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
```

Немедленно после того, как операция `new` создала объект, вызывается *конструктор* объекта для выполнения инициализации. Конструктор определяется подобно методу за исключением того, что вместо имени метода и возвращаемого типа указывается имя типа, к которому относится конструктор:

```
public class UnitConverter
{
  ...
  public UnitConverter (int unitRatio) { ratio = unitRatio; }
  ...
}
```

### Члены экземпляра и статические члены

Данные-члены и функции-члены, которые оперируют на *экземпляре* типа, называются членами экземпляра. Примерами членов экземпляра могут служить метод `Convert` в типе `UnitConverter` и метод `ToString` в типе `int`. По умолчанию члены являются членами экземпляра.

Данные-члены и функции-члены, которые не оперируют на экземпляре типа, а взамен имеют дело с самим типом, должны помечаться как статические (`static`). Примерами статических методов являются `Test.Main` и `Console.WriteLine`. Класс `Console` в действительности — это *статический класс*, из чего следует, что *все* его члены статические. Создавать экземпляры класса `Console` никогда не придется — одна консоль разделяется всем приложением.

Давайте сравним члены экземпляра и статические члены. В следующем коде поле экземпляра `Name` принадлежит экземпляру `Panda`, тогда как поле `Population` относится к набору всех экземпляров класса `Panda`:

```
public class Panda
{
  public string Name;               // Поле экземпляра
  public static int Population;     // Статическое поле
  public Panda (string n)           // Конструктор
  {
    Name = n;                       // Присвоить значение полю экземпляра
    Population = Population + 1; // Инкрементировать значение статического поля
  }
}
```

В показанном ниже коде создаются два экземпляра `Panda`, а затем выводятся их имена (поле `Name`) и общее количество (поле `Population`):

```
using System;

class Test
{
  static void Main()
  {
    Panda p1 = new Panda ("Pan Dee");
    Panda p2 = new Panda ("Pan Dah");

    Console.WriteLine (p1.Name);  // Pan Dee
    Console.WriteLine (p2.Name);  // Pan Dah

    Console.WriteLine (Panda.Population);  // 2
  }
}
```

Попытка использования `p1.Population` или `Panda.Name` приведет к возникновению ошибки на этапе компиляции.

### Ключевое слово `public`

Ключевое слово `public` открывает доступ к членам со стороны других классов. Если бы в рассматриваемом примере поле `Name` класса `Panda` не было помечено как `public`, то оно стало бы закрытым, и класс `Test` не сумел бы получить к нему доступ. Маркировка члена как открытого (`public`) означает, что данный тип разрешает другим типам видеть этот член, а все остальное будет относиться к закрытым деталям реализации. В рамках объектно-ориентированной терминологии говорят, что открытые члены *инкапсулируют* закрытые члены класса.

## Преобразования

В C# возможны преобразования между экземплярами совместимых типов. Преобразование всегда создает новое значение из существующего. Преобразования могут быть либо *неявными*, либо *явными*: неявные преобразования происходят автоматически, в то время как явные преобразования требуют *приведения*. В следующем примере мы *неявно* преобразуем тип `int` в `long` (который имеет в два раза больше битов, чем `int`) и *явно* приводим тип `int` к `short` (который имеет вполовину меньше битов, чем `int`):

```
int x = 12345;          // int - 32-битное целое
long y = x;             // Неявное преобразование в 64-битное целое
short z = (short)x;     // Явное преобразование в 16-битное целое
```

Неявные преобразования разрешены, когда удовлетворяются перечисленные ниже условия:

- компилятор может гарантировать, что они всегда будут проходить успешно;
- в результате преобразования никакая информация не утрачивается[1].

И наоборот, явные преобразования требуются, когда справедливо одно из следующих утверждений:

- компилятор не может гарантировать, что они всегда будут проходить успешно;
- в результате преобразования информация может быть утрачена.

(Если компилятор может определить, что преобразование будет *всегда* терпеть неудачу, то оба вида преобразования запрещаются. Преобразования, в которых участвуют обобщения, в определенных обстоятельствах также могут потерпеть неудачу; об этом пойдет речь в разделе "Параметры типа и преобразования" главы 3.)

*Числовые преобразования*, которые мы только что видели, встроены в язык. Вдобавок в C# поддерживаются *ссылочные преобразования* и *упаковывающие преобразования* (см. главу 3), а также *специальные преобразования* (см. раздел "Перегрузка операций" в главе 4). Компилятор не навязывает упомянутые выше правила для специальных преобразований, поэтому неудачно спроектированные типы могут вести себя по-другому.

## Типы значений и ссылочные типы

Все типы C# делятся на следующие категории:

- типы значений;
- ссылочные типы;
- параметры типа в обобщениях;
- типы указателей.

В этом разделе мы опишем типы значений и ссылочные типы. Параметры типа в обобщениях будут рассматриваться в разделе "Обобщения" главы 3, а типы указателей – в разделе "Небезопасный код и указатели" главы 4.

[1] Небольшое предостережение: очень высокие значения `long` после преобразования в `double` теряют в точности.

*Типы значений* включают большинство встроенных типов (а именно – все числовые типы, тип `char` и тип `bool`), а также специальные типы `struct` и `enum`.

*Ссылочные типы* включают все классы, массивы, делегаты и интерфейсы. (Сюда также входит предопределенный тип `string`.)

Фундаментальное отличие между типами значений и ссылочными типами связано с тем, каким образом они хранятся в памяти.

## Типы значений

Содержимым переменной или константы, относящейся к *типу значения*, является просто значение. Например, содержимое встроенного типа значения `int` – 32 бита данных.

С помощью ключевого слова `struct` можно определить специальный тип значения (рис. 2.1):

```
public struct Point { public int X; public int Y; }
```

или более лаконично:

```
public struct Point { public int X, Y; }
```

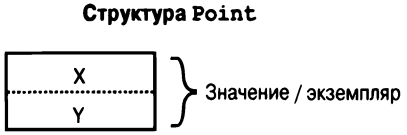



***Рис. 2.1.** Экземпляр типа значения в памяти*

Присваивание экземпляра типа значения всегда приводит к *копированию* данного экземпляра. Например:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;           // Присваивание приводит к копированию

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                // Изменить p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 7
}
```

На рис. 2.2 видно, что экземпляры `p1` и `p2` хранятся независимо друг от друга.


***Рис. 2.2.** Присваивание копирует экземпляр типа значения*

## Ссылочные типы

Ссылочный тип сложнее типа значения из-за наличия двух частей: *объекта* и *ссылки* на этот объект. Содержимым переменной или константы ссылочного типа является ссылка на объект, который содержит значение. Ниже приведен тип `Point` из предыдущего примера, переписанный в виде класса (рис. 2.3):

```
public class Point { public int X, Y; }
```

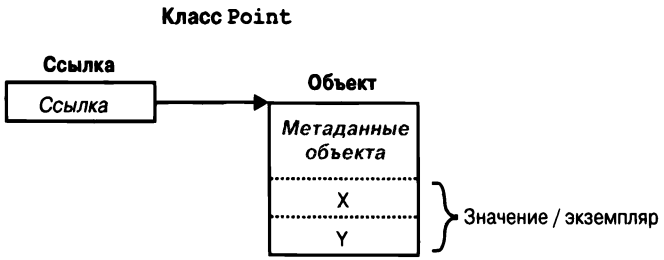



*Рис. 2.3. Экземпляр ссылочного типа в памяти*

Присваивание переменной ссылочного типа вызывает копирование ссылки, но не экземпляра объекта. В результате множество переменных могут ссылаться на один и тот же объект — то, что обычно невозможно с типами значений. Если повторить предыдущий пример при условии, что `Point` теперь является классом, операция над `p1` будет воздействовать на `p2`:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;           // Копирует ссылку на p1

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                // Изменить p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 9
```

На рис. 2.4 видно, что `p1` и `p2` — две ссылки, указывающие на один и тот же объект.

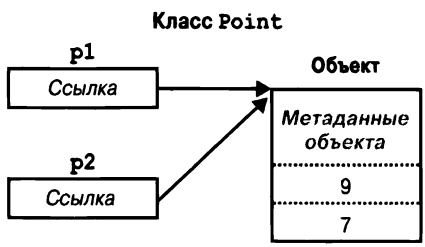



*Рис. 2.4. Присваивание копирует ссылку*

### Значение null

Ссылке может быть присвоен литерал null, который отражает тот факт, что ссылка не указывает на какой-либо объект:

```
class Point {...}
...
Point p = null;
Console.WriteLine (p == null);   // True

// Следующая строка вызывает ошибку времени выполнения
// (генерируется исключение NullReferenceException):
Console.WriteLine (p.X);
```

Напротив, тип значения обычно не может иметь значение null:

```
struct Point {...}
...
Point p = null;       // Ошибка на этапе компиляции
int x = null;         // Ошибка на этапе компиляции
```

В C# также имеется специальная конструкция под названием *типы, допускающие значение* null, которая предназначена для представления null в типах значений (см. раздел "Типы, допускающие значение null" в главе 4).

### Накладные расходы, связанные с хранением

Экземпляры типов значений занимают в точности столько памяти, сколько требуется для хранения их полей. В рассматриваемом примере Point требует 8 байтов памяти:

```
struct Point
{
  int x;  // 4 байта
  int y;  // 4 байта
}
```

Формально среда CLR располагает поля внутри типа по адресу, кратному размеру полей (выровненному максимум до 8 байтов). Таким образом, следующая структура в действительности потребляет 16 байтов памяти (с семью байтами после первого поля, которые "тратятся впустую"):

```
struct A { byte b; long l; }
```

Это поведение можно переопределить с помощью атрибута StructLayout (см. раздел "Отображение структуры на неуправляемую память" в главе 25).

Ссылочные типы требуют раздельного выделения памяти для ссылки и объекта. Объект потребляет столько памяти, сколько необходимо его полям, плюс дополнительный объем на административные нужды. Точный объем накладных расходов по существу зависит от реализации исполняющей среды .NET, но составляет минимум восемь байтов, которые применяются для хранения ключа к типу объекта, а также временной информации, такой как его состояние блокировки для многопоточной обработки и флаг для указания, был ли объект закреплен, чтобы он не перемещался сборщиком мусора. Каждая ссылка на объект требует дополнительных четырех или восьми байтов в зависимости от того, на какой платформе функционирует исполняющая среда .NET – 32- или 64-разрядной.

## Классификация предопределенных типов

Предопределенные типы в C# классифицируются следующим образом.

*Типы значений*

- Числовой
  - Целочисленный со знаком (`sbyte`, `short`, `int`, `long`)
  - Целочисленный без знака (`byte`, `ushort`, `uint`, `ulong`)
  - Вещественный (`float`, `double`, `decimal`)
- Логический (`bool`)
- Символьный (`char`)

*Ссылочные типы*

- Строка (`string`)
- Объект (`object`)

Предопределенные типы C# являются псевдонимами типов .NET Framework в пространстве имен `System`. Показанные ниже два оператора отличаются только синтаксисом:

```
int i = 5;
System.Int32 i = 5;
```

Набор предопределенных типов *значений*, исключая `decimal`, известен в CLR как *примитивные типы*. Примитивные типы называются так потому, что они поддерживаются непосредственно через инструкции в скомпилированном коде, которые обычно транслируются в прямую поддержку внутри имеющегося процессора. Например:

```
                  // Лежащие в основе шестнадцатеричные представления
int i = 7;        // 0x7
bool b = true;    // 0x1
char c = 'A';     // 0x41
float f = 0.5f; // Использует кодирование чисел с плавающей точкой IEEE
```

Типы `System.IntPtr` и `System.UIntPtr` также относятся к примитивным (см. главу 25).

# Числовые типы

Предопределенные числовые типы C# показаны в табл. 2.1.

**Таблица 2.1. Предопределенные числовые типы в C#**

| Тип C# | Тип в пространстве имен `System` | Суффикс | Размер в битах | Диапазон |
|---|---|---|---|---|
| **Целочисленный со знаком** | | | | |
| `sbyte` | `SByte` | | 8 | $-2^7$ — $2^7-1$ |
| `short` | `Int16` | | 16 | $-2^{15}$ — $2^{15}-1$ |
| `int` | `Int32` | | 32 | $-2^{31}$ — $2^{31}-1$ |
| `long` | `Int64` | `L` | 64 | $-2^{63}$ — $2^{63}-1$ |

*Окончание табл. 2.1*

| Тип C# | Тип в пространстве имен System | Суффикс | Размер в битах | Диапазон |
|---|---|---|---|---|
| **Целочисленный без знака** | | | | |
| byte | Byte | | 8 | $0 — 2^8-1$ |
| ushort | UInt16 | | 16 | $0 — 2^{16}-1$ |
| uint | UInt32 | U | 32 | $0 — 2^{32}-1$ |
| ulong | UInt64 | UL | 64 | $0 — 2^{64}-1$ |
| **Вещественный** | | | | |
| float | Single | F | 32 | $\pm(\sim10^{-45} — 10^{38})$ |
| double | Double | D | 64 | $\pm(\sim10^{-324} — 10^{308})$ |
| decimal | Decimal | M | 128 | $\pm(\sim10^{-28} — 10^{28})$ |

Из всех *целочисленных* типов int и long являются первоклассными типами, которым обеспечивается поддержка как в языке C#, так и в исполняющей среде. Другие целочисленные типы обычно применяются для реализации взаимодействия или когда главная задача связана с эффективностью хранения.

В рамках вещественных числовых типов float и double называются *типами с плавающей точкой*[2] и обычно используются в научных и графических вычислениях. Тип decimal, как правило, применяется в финансовых вычислениях, где требуется десятичная арифметика и высокая точность.

## Числовые литералы

*Целочисленные литералы* могут использовать десятичную или шестнадцатеричную форму записи; шестнадцатеричная форма записи предусматривает применение префикса 0x. Например:

```
int x = 127;
long y = 0x7F;
```

В версии C# 7 разрешено вставлять символы подчеркивания где угодно внутри числового литерала, делая его более читабельным:

```
int million = 1_000_000;
```

В C# 7 также можно указывать числа в двоичном виде с помощью префикса 0b:

```
var b = 0b1010_1011_1100_1101_1110_1111;
```

Вещественные литералы могут использовать десятичную и/или экспоненциальную форму записи. Например:

```
double d = 1.5;
double million = 1E06;
```

[2] Формально decimal — тоже тип с плавающей точкой, хотя в спецификации языка C# он не упоминается в таком качестве.

### Выведение типа числового литерала

По умолчанию компилятор *выводит* тип числового литерала, относя его либо к `double`, либо к какому-то целочисленному типу.

Если литерал содержит десятичную точку или символ экспоненты (`E`), то он получает тип `double`.

В противном случае типом литерала будет первый тип, способный уместить значение литерала, из следующего списка: `int`, `uint`, `long` и `ulong`.

Например:

```
Console.WriteLine (        1.0.GetType());      // Double  (double)
Console.WriteLine (       1E06.GetType());      // Double  (double)
Console.WriteLine (          1.GetType());      // Int32   (int)
Console.WriteLine ( 0xF0000000.GetType());      // UInt32  (uint)
Console.WriteLine (0x100000000.GetType());      // Int64   (long)
```

### Числовые суффиксы

*Числовые суффиксы* явно определяют тип литерала. Суффиксы могут записываться либо строчными, либо прописными буквами; все они перечислены ниже.

| Суффикс | Тип C# | Пример |
|---|---|---|
| F | float | `float f = 1.0F;` |
| D | double | `double d = 1D;` |
| M | decimal | `decimal d = 1.0M;` |
| U | uint | `uint i = 1U;` |
| L | long | `long i = 1L;` |
| UL | ulong | `ulong i = 1UL;` |

Необходимость в суффиксах `U` и `L` возникает редко, поскольку типы `uint`, `long` и `ulong` могут почти всегда или *выводиться*, или *неявно преобразовываться* из `int`:

```
long i = 5;  // Неявное преобразование без потерь литерала int в тип long
```

Суффикс `D` формально является избыточным из-за того, что все литералы с десятичной точкой выводятся в тип `double`. И к числовому литералу всегда можно добавить десятичную точку:

```
double x = 4.0;
```

Суффиксы `F` и `M` наиболее полезны и всегда должны применяться при указании литералов `float` или `decimal`. Без суффикса `F` следующая строка кода не скомпилируется, т.к. значение `4.5` выводится в тип `double`, для которого не существует неявного преобразования в тип `float`:

```
float f = 4.5F;
```

Тот же принцип справедлив для десятичных литералов:

```
decimal d = -1.23M;  // Не скомпилируется без суффикса M
```

Семантика числовых преобразований подробно описана в следующем разделе.

# Числовые преобразования

## Преобразования между целочисленными типами

Преобразования между целочисленными типами являются *неявными*, когда целевой тип в состоянии представить каждое возможное значение исходного типа. В противном случае требуется *явное* преобразование. Например:

```
int x = 12345;       // int - 32-битный целочисленный тип
long y = x;          // Неявное преобразование в 64-битный целочисленный тип
short z = (short)x; // Явное преобразование в 16-битный целочисленный тип
```

## Преобразования между типами с плавающей точкой

Тип `float` может быть неявно преобразован в `double`, т.к. `double` позволяет представить любое возможное значение `float`. Обратное преобразование должно быть явным.

## Преобразования между типами с плавающей точкой и целочисленными типами

Все целочисленные типы могут быть неявно преобразованы во все типы с плавающей точкой:

```
int i = 1;
float f = i;
```

Обратное преобразование обязано быть явным:

```
int i2 = (int)f;
```

Когда число с плавающей точкой приводится к целому, любая дробная часть отбрасывается; никакого округления не производится. Статический класс `System.Convert` предоставляет методы, которые выполняют преобразования между разнообразными числовыми типами с округлением (см. главу 6).

Неявное преобразование большого целочисленного типа в тип с плавающей точкой сохраняет *величину*, но иногда может приводить к потере *точности*. Причина в том, что типы с плавающей точкой всегда имеют большую величину, чем целочисленные типы, но могут иметь меньшую точность. Для демонстрации сказанного рассмотрим пример с большим числом:

```
int i1 = 100000001;
float f = i1;      // Величина сохраняется, точность теряется
int i2 = (int)f;  // 100000000
```

## Десятичные преобразования

Все целочисленные типы могут быть неявно преобразованы в `decimal`, поскольку тип `decimal` способен представлять любое возможное целочисленное значение в C#. Все остальные числовые преобразования в и из типа `decimal` должны быть явными.

# Арифметические операции

Арифметические операции (`+`, `-`, `*`, `/`, `%`) определены для всех числовых типов, исключая 8- и 16-битные целочисленные типы:

- + Сложение
- - Вычитание
- * Умножение
- / Деление
- % Остаток от деления

## Операции инкремента и декремента

Операции инкремента и декремента (++, --) увеличивают и уменьшают значения переменных числовых типов на 1. Эти операции могут находиться перед или после имени переменной в зависимости от того, когда требуется обновить значение переменной — *до* или *после* вычисления выражения. Например:

```
int x = 0, y = 0;
Console.WriteLine (x++);  // Выводит 0; x теперь содержит 1
Console.WriteLine (++y);  // Выводит 1; y теперь содержит 1
```

## Специальные операции с целочисленными типами

(К *целочисленным типам* относятся `int`, `uint`, `long`, `ulong`, `short`, `ushort`, `byte` и `sbyte`.)

### Деление

Операции деления с целочисленными типами всегда усекают остаток (округляют в направлении нуля). Деление на переменную, значение которой равно нулю, вызывает ошибку во время выполнения (исключение `DivideByZeroException`):

```
int a = 2 / 3;  // 0

int b = 0;
int c = 5 / b;  // Генерируется исключение DivideByZeroException
```

Деление на *литерал* или *константу* 0 генерирует ошибку на этапе компиляции.

### Переполнение

Во время выполнения арифметические операции с целочисленными типами могут привести к переполнению. По умолчанию это происходит молча — никакие исключения не генерируются, а результат демонстрирует поведение с циклическим возвратом, как если бы вычисление производилось над бо́льшим целочисленным типом с отбрасыванием дополнительных значащих битов. Например, декрементирование минимально возможного значения типа `int` дает в результате максимально возможное значение `int`:

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue);  // True
```

### Операции проверки переполнения

Операция `checked` сообщает исполняющей среде о том, что вместо молчаливого переполнения она должна генерировать исключение `OverflowException`, когда выражение или оператор над целочисленным типом выходит за арифметические пределы этого типа. Операция `checked` воздействует на выражения с операциями ++, --, +, - (бинарной и унарной), *, / и явными преобразованиями между целочисленными типами.

Операция checked не оказывает никакого влияния на типы double и float (которые получают при переполнении специальные значения "бесконечности", как вскоре будет показано) и на тип decimal (который проверяется всегда).

Операцию checked можно использовать либо с выражением, либо с блоком операторов. Например:

```
int a = 1000000;
int b = 1000000;
int c = checked (a * b);  // Проверяет только это выражение
checked                   // Проверяет все выражения
{                         // в блоке операторов
  ...
  c = a * b;
  ...
}
```

Проверку на арифметическое переполнение можно сделать обязательной для всех выражений в программе, скомпилировав ее с переключателем командной строки /checked+ (в Visual Studio это делается на вкладке Advanced Build Settings (Дополнительные параметры сборки)). Если позже понадобится отключить проверку переполнения для конкретных выражений или операторов, тогда можно воспользоваться операцией unchecked. Например, следующий код не будет генерировать исключения, даже если его скомпилировать с переключателем /checked+:

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

## Проверка переполнения для константных выражений

Независимо от наличия переключателя /checked компилятора для выражений, оцениваемых во время компиляции, проверка переполнения производится всегда, если только не применена операция unchecked:

```
int x = int.MaxValue + 1;                // Ошибка на этапе компиляции
int y = unchecked (int.MaxValue + 1);    // Ошибки отсутствуют
```

## Побитовые операции

В C# поддерживаются следующие побитовые операции.

| Операция | Описание | Пример выражения | Результат |
|---|---|---|---|
| ~ | Дополнение | ~0xfU | 0xfffffff0U |
| & | И | 0xf0 & 0x33 | 0x30 |
| \| | ИЛИ | 0xf0 \| 0x33 | 0xf3 |
| ^ | Исключающее ИЛИ | 0xff00 ^ 0x0ff0 | 0xf0f0 |
| << | Сдвиг влево | 0x20 << 2 | 0x80 |
| >> | Сдвиг вправо | 0x20 >> 1 | 0x10 |

## 8- и 16-битные целочисленные типы

К 8- и 16-битным целочисленным типам относятся `byte`, `sbyte`, `short` и `ushort`. У указанных типов отсутствуют собственные арифметические операции, поэтому в C# они при необходимости неявно преобразуются в более крупные типы. Попытка присваивания результата переменной меньшего целочисленного типа может привести к получению ошибки на этапе компиляции:

```
short x = 1, y = 1;
short z = x + y;              // Ошибка на этапе компиляции
```

В данном случае переменные `x` и `y` неявно преобразуются в тип `int`, поэтому сложение может быть выполнено. Это означает, что результат также будет иметь тип `int`, который не может быть неявно приведен к типу `short` (из-за возможной потери информации). Чтобы такой код скомпилировался, потребуется добавить явное приведение:

```
short z = (short) (x + y);      // Компилируется
```

## Специальные значения `float` и `double`

В отличие от целочисленных типов типы с плавающей точкой имеют значения, которые определенные операции трактуют особым образом. Такими специальными значениями являются NaN (Not a Number – не число), +∞, −∞ и -0. В классах `float` и `double` предусмотрены константы для NaN, +∞ и −∞, а также для других значений (`MaxValue`, `MinValue` и `Epsilon`). Например:

```
Console.WriteLine (double.NegativeInfinity);  // Минус бесконечность
```

Ниже перечислены константы, которые представляют специальные значения для типов `double` и `float`.

| Специальное значение | Константа `double` | Константа `float` |
|---|---|---|
| NaN | `double.NaN` | `float.NaN` |
| +∞ | `double.PositiveInfinity` | `float.PositiveInfinity` |
| −∞ | `double.NegativeInfinity` | `float.NegativeInfinity` |
| -0 | `-0.0` | `-0.0f` |

Деление ненулевого числа на ноль дает в результате бесконечную величину. Например:

```
Console.WriteLine ( 1.0 /  0.0);  // Бесконечность
Console.WriteLine (-1.0 /  0.0);  // Минус бесконечность
Console.WriteLine ( 1.0 / -0.0);  // Минус бесконечность
Console.WriteLine (-1.0 / -0.0);  // Бесконечность
```

Деление нуля на ноль или вычитание бесконечности из бесконечности дает в результате NaN. Например:

```
Console.WriteLine ( 0.0 /  0.0);                 //  NaN
Console.WriteLine ((1.0 /  0.0) - (1.0 / 0.0));  //  NaN
```

Когда применяется операция ==, значение NaN никогда не будет равно другому значению, даже NaN:

```
Console.WriteLine (0.0 / 0.0 == double.NaN);     // False
```

Для проверки, является ли значение специальным значением NaN, должен использоваться метод `float.IsNaN` или `double.IsNaN`:

```
Console.WriteLine (double.IsNaN (0.0 / 0.0));    // True
```

Однако в случае применения метода `object.Equals` два значения NaN равны:

```
Console.WriteLine (object.Equals (0.0 / 0.0, double.NaN)); // True
```

Значения NaN иногда удобны для представления специальных величин. Например, в WPF с помощью `double.NaN` представлено измерение, значением которого является "Automatic" (автоматическое). Другой способ представления такого значения предусматривает использование типа, допускающего значение `null` (глава 4), а еще один способ – применение специальной структуры, которая является оболочкой для числового типа с дополнительным полем (глава 3).

Типы `float` и `double` следуют спецификации IEEE 754 для формата представления чисел с плавающей точкой, поддерживаемой практически всеми процессорами. Подробную информацию относительно поведения этих типов можно найти на вебсайте `http://www.ieee.org`.

## Выбор между `double` и `decimal`

Тип `double` удобен в научных вычислениях (таких как расчет пространственных координат), а тип `decimal` – в финансовых вычислениях и для представления значений, которые являются "искусственными", а не полученными в результате реальных измерений. Ниже представлен обзор отличий между типами `double` и `decimal`.

| Характеристика | `double` | `decimal` |
|---|---|---|
| Внутреннее представление | Двоичное | Десятичное |
| Десятичная точность | 15–16 значащих цифр | 28–29 значащих цифр |
| Диапазон | $\pm(\sim10^{-324}$ — $\sim10^{308})$ | $\pm(\sim10^{-28}$ — $\sim10^{28})$ |
| Специальные значения | +0, –0, +∞, –∞ и NaN | Отсутствуют |
| Скорость обработки | Присущая процессору | Не присущая процессору (примерно в 10 раз медленнее, чем в случае `double`) |

## Ошибки округления вещественных чисел

Типы `float` и `double` внутренне представляют числа в двоичной форме. По этой причине точно представляются только числа, которые могут быть выражены в двоичной системе счисления. На практике это означает, что большинство литералов с дробной частью (которые являются десятичными) не будут представлены точно. Например:

```
float tenth = 0.1f;                    // Не точно 0.1
float one   = 1f;
Console.WriteLine (one - tenth * 10f);  // -1.490116E-08
```

Именно потому типы `float` и `double` не подходят для финансовых вычислений. В противоположность им тип `decimal` работает в десятичной системе счисления, так

что он способен точно представлять дробные числа вроде 0.1, выразимые в десятичной системе (а также в системах счисления с основаниями-множителями 10 – двоичной и пятеричной). Поскольку вещественные литералы являются десятичными, тип decimal может точно представлять такие числа, как 0.1. Тем не менее, ни double, ни decimal не могут точно представить дробное число, десятичное представление которого является периодическим:

```
decimal m = 1M / 6M;    // 0.16666666666666666666666666667M
double  d = 1.0 / 6.0;  // 0.16666666666666666
```

Это приводит к накапливающимся ошибкам округления:

```
decimal notQuiteWholeM = m+m+m+m+m+m;  // 1.00000000000000000000000002M
double  notQuiteWholeD = d+d+d+d+d+d;  // 0.99999999999999989
```

которые нарушают работу операций эквивалентности и сравнения:

```
Console.WriteLine (notQuiteWholeM == 1M);  // False
Console.WriteLine (notQuiteWholeD < 1.0);  // True
```

# Булевские типы и операции

Тип bool в C# (псевдоним типа System.Boolean) представляет логическое значение, которому может быть присвоен литерал true или false.

Хотя для хранения булевского значения достаточно только одного бита, исполняющая среда будет использовать один байт памяти, т.к. это минимальная порция, с которой исполняющая среда и процессор могут эффективно работать. Во избежание непродуктивных расходов пространства в случае массивов платформа .NET Framework предлагает в пространстве имен System.Collections класс BitArray, который позволяет задействовать по одному биту для каждого булевского значения в массиве.

## Булевские преобразования

Приведения и преобразования из типа bool в числовые типы и наоборот не разрешены.

## Операции сравнения и проверки равенства

Операции == и != проверяют на предмет эквивалентности и неэквивалентности значения любого типа и всегда возвращают значение bool[3]. Типы значений обычно поддерживают очень простое понятие эквивалентности:

```
int x = 1;
int y = 2;
int z = 1;
Console.WriteLine (x == y);  // False
Console.WriteLine (x == z);  // True
```

Для ссылочных типов эквивалентность по умолчанию основана на *ссылке*, а не на действительном *значении* лежащего в основе объекта (более подробно об этом рассказывается в главе 6):

[3] Эти операции допускается *перегружать* (глава 4), чтобы они возвращали тип, отличный от bool, но на практике так почти никогда не поступают.

```
public class Dude
{
  public string Name;
  public Dude (string n) { Name = n; }
}
...
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2);  // False
Dude d3 = d1;
Console.WriteLine (d1 == d3);  // True
```

Операции эквивалентности и сравнения, ==, !=, <, >, >= и <=, работают со всеми числовыми типами, но должны осмотрительно применяться с вещественными числами (как было указано выше в разделе "Ошибки округления вещественных чисел"). Операции сравнения также работают с членами типа `enum`, сравнивая лежащие в их основе целочисленные значения. Это будет описано в разделе "Перечисления" главы 3.

Операции эквивалентности и сравнения более подробно объясняются в разделе "Перегрузка операций" главы 4, а также в разделах "Сравнение эквивалентности" и "Сравнение порядка" главы 6.

## Условные операции

Операции `&&` и `||` реализуют условия *И* и *ИЛИ*. Они часто применяются в сочетании с операцией `!`, которая выражает условие *НЕ*. В показанном ниже примере метод `UseUmbrella` (брать ли зонт) возвращает `true`, если дождливо (`rainy`) или солнечно (`sunny`) при условии, что также не ветрено (`windy`):

```
static bool UseUmbrella (bool rainy, bool sunny, bool windy)
{
  return !windy && (rainy || sunny);
}
```

Когда возможно, операции `&&` и `||` *сокращают* вычисления. В предыдущем примере, если ветрено (`windy`), тогда выражение `(rainy || sunny)` даже не оценивается. Сокращение вычислений играет важную роль в обеспечении выполнения выражений, таких как показанное ниже, без генерации исключения `NullReferenceException`:

```
if (sb != null && sb.Length > 0) ...
```

Операции `&` и `|` также реализуют условия *И* и *ИЛИ*:

```
return !windy & (rainy | sunny);
```

Их отличие состоит в том, что они *не сокращают вычисления*. По этой причине `&` и `|` редко используются в качестве операций сравнения.

В отличие от языков C и C++ операции `&` и `|` производят *булевские* сравнения (без сокращения вычислений), когда применяются к выражениям `bool`. Операции `&` и `|` выполняются как побитовые только в случае применения к числам.

### Условная (тернарная) операция

*Условная операция* (чаще называемая *тернарной операцией*, т.к. она единственная принимает три операнда) имеет вид `q ? a : b`, где результатом является `a`, если условие `q` равно `true`, и `b` — в противном случае. Например:

```
static int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

Условная операция особенно удобна в запросах LINQ (глава 8).

## Строки и символы

Тип `char` в C# (псевдоним типа `System.Char`) представляет символ Unicode и занимает 2 байта. Литерал `char` указывается в одинарных кавычках:

```
char c = 'A';  // Простой символ
```

*Управляющие последовательности* выражают символы, которые не могут быть представлены или интерпретированы буквально. Управляющая последовательность состоит из символа обратной косой черты, за которым следует символ со специальным смыслом. Например:

```
char newLine = '\n';
char backSlash = '\\';
```

Символы управляющих последовательностей показаны в табл. 2.2.

**Таблица 2.2. Символы управляющих последовательностей**

| Символ | Смысл | Значение |
|---|---|---|
| `\'` | Одинарная кавычка | 0x0027 |
| `\"` | Двойная кавычка | 0x0022 |
| `\\` | Обратная косая черта | 0x005C |
| `\0` | Пусто | 0x0000 |
| `\a` | Сигнал тревоги | 0x0007 |
| `\b` | Забой | 0x0008 |
| `\f` | Перевод страницы | 0x000C |
| `\n` | Новая строка | 0x000A |
| `\r` | Возврат каретки | 0x000D |
| `\t` | Горизонтальная табуляция | 0x0009 |
| `\v` | Вертикальная табуляция | 0x000B |

Управляющая последовательность `\u` (или `\x`) позволяет указывать любой символ Unicode в виде его шестнадцатеричного кода, состоящего из четырех цифр:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

## Символьные преобразования

Неявное преобразование `char` в числовой тип работает для числовых типов, которые могут вместить значение `short` без знака. Для других числовых типов требуется явное преобразование.

## Строковый тип

Тип `string` в C# (псевдоним типа `System.String`, подробно рассматриваемый в главе 6) представляет неизменяемую последовательность символов Unicode. Строковый литерал указывается в двойных кавычках:

```
string a = "Heat";
```

`string` — это ссылочный тип, а не тип значения. Тем не менее, его операции эквивалентности следуют семантике типов значений:

```
string a = "test";
string b = "test";
Console.Write (a == b);  // True
```

Управляющие последовательности, допустимые для литералов `char`, также работают внутри строк:

```
string a = "Here's a tab:\t";
```

Платой за это является необходимость дублирования символа обратной косой черты, когда он нужен буквально:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

Чтобы избежать такой проблемы, в C# разрешены *дословные* строковые литералы. Дословный строковый литерал снабжается префиксом `@` и не поддерживает управляющие последовательности. Следующая дословная строка идентична предыдущей строке:

```
string a2 = @"\\server\fileshare\helloworld.cs";
```

Дословный строковый литерал может также занимать несколько строк:

```
string escaped  = "First Line\r\nSecond Line";
string verbatim = @"First Line
Second Line";

// Выводит True, если в IDE-среде используются разделители строк CR-LF:
Console.WriteLine (escaped == verbatim);
```

Чтобы включить в дословный строковый литерал символ двойной кавычки, его понадобится записать дважды:

```
string xml = @"<customer id=""123""></customer>";
```

### Конкатенация строк

Операция + выполняет конкатенацию двух строк:

```
string s = "a" + "b";
```

Один из операндов может быть нестроковым значением; в этом случае для него будет вызван метод `ToString`. Например:

```
string s = "a" + 5;  // a5
```

Многократное применение операции + для построения строки является неэффективным: более удачное решение предусматривает использование типа `System.Text.StringBuilder` (описанного в главе 6).

### Интерполяция строк (C# 6)

Строка, предваренная символом `$`, называется *интерполированной строкой*. Интерполированные строки могут содержать выражения, заключенные в фигурные скобки:

```
int x = 4;
Console.Write ($"A square has {x} sides");  // Выводит: A square has 4 sides
```

Внутри скобок может быть указано любое допустимое выражение C# произвольного типа, и C# преобразует это выражение в строку, вызывая `ToString` или эквивалентный метод данного типа. Форматирование можно изменять путем добавления к выражению двоеточия и *форматной строки* (форматные строки описаны в разделе "Форматирование и разбор" главы 6):

```
string s = $"255 in hex is {byte.MaxValue:X2}";  // X2 - шестнадцатеричное
                                                 // значение из двух цифр
                                                 // s получает значение "255 in hex is FF"
```

Интерполированные строки должны находиться в одной строке кода, если только вы также не укажете операцию дословной строки. Обратите внимание, что операция `$` должна располагаться перед `@`:

```
int x = 2;
string s = $@"this spans {
x} lines";
```

Для включения в интерполированную строку литеральной фигурной скобки символ фигурной скобки должен быть продублирован.

### Сравнения строк

Тип `string` не поддерживает операции < и > для сравнений. Вместо них должен применяться метод `CompareTo` типа `string`, который рассматривается в главе 6.

## Массивы

Массив представляет фиксированное количество переменных (называемых *элементами*) определенного типа. Элементы массива всегда хранятся в непрерывном блоке памяти, обеспечивая высокоэффективный доступ.

Массив обозначается квадратными скобками после типа элементов. Например:

```
char[] vowels = new char[5];  // Объявить массив из 5 символов
```

С помощью квадратных скобок также указывается *индекс* в массиве, что позволяет получать доступ к элементам по их позициям:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine (vowels[1]);  // e
```

Код приведет к выводу буквы "e", поскольку массив индексируется, начиная с 0. Оператор цикла `for` можно использовать для прохода по всем элементам в массиве. Цикл `for` в следующем примере выполняется для целочисленных значений `i` от 0 до 4:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels[i]);          // aeiou
```

Свойство `Length` массива возвращает количество элементов в массиве. После создания массива изменить его длину невозможно. Пространство имен `System.Collection` и вложенные в него пространства имен предоставляют такие высокоуровневые структуры данных, как массивы с динамически изменяемыми размерами и словари.

*Выражение инициализации массива* позволяет объявлять и заполнять массив в единственном операторе:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

или проще:

```
char[] vowels = {'a','e','i','o','u'};
```

Все массивы унаследованы от класса `System.Array`, который предоставляет общие службы для всех массивов. В состав его членов входят методы для получения и установки элементов независимо от типа массива; они описаны в разделе "Класс `Array`" главы 7.

## Стандартная инициализация элементов

При создании массива всегда происходит инициализация его элементов стандартными значениями. Стандартное значение для типа представляет собой результат побитового обнуления памяти. Например, пусть создается массив целых чисел. Поскольку `int` – тип значения, выделится пространство под 1000 целочисленных значений в непрерывном блоке памяти. Стандартным значением для каждого элемента будет 0:

```
int[] a = new int[1000];
Console.Write (a[123]);  // 0
```

### Сравнение типов значений и ссылочных типов

Значительное влияние на производительность оказывает то, какой тип имеют элементы массива – тип значения или ссылочный тип. Если элементы относятся к типу значения, то пространство под значение каждого элемента выделяется как часть массива. Например:

```
public struct Point { public int X, Y; }
...
Point[] a = new Point[1000];
int x = a[500].X;  // 0
```

Если бы `Point` был классом, тогда создание массива привело бы просто к выделению пространства под 1000 ссылок `null`:

```
public class Point { public int X, Y; }
...
Point[] a = new Point[1000];
int x = a[500].X; // Ошибка во время выполнения, исключение NullReferenceException
```

Чтобы устранить ошибку, после создания экземпляра массива потребуется явно создать 1000 экземпляров `Point`:

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++)  // Цикл для i от 0 до 999
  a[i] = new Point();               // Установить i-й элемент массива
                                    // в новый экземпляр Point
```

Независимо от типа элементов массив *сам по себе* всегда является объектом ссылочного типа. Например, следующий оператор допустим:

```
int[] a = null;
```

## Многомерные массивы

Многомерные массивы бывают двух видов: *прямоугольные* и *зубчатые*. Прямоугольный массив представляет *n*-мерный блок памяти, а зубчатый массив — это массив, содержащий массивы.

### Прямоугольные массивы

Прямоугольные массивы объявляются с применением запятых для отделения каждого измерения друг от друга. Ниже приведено объявление прямоугольного двумерного массива размерностью 3×3:

```
int[,] matrix = new int[3,3];
```

Метод `GetLength` массива возвращает длину для заданного измерения (начиная с 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix[i,j] = i * 3 + j;
```

Прямоугольный массив может быть инициализирован следующим образом (здесь создается массив, идентичный предыдущему примеру):

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

### Зубчатые массивы

Зубчатые массивы объявляются с использованием последовательно идущих пар квадратных скобок, которые представляют каждое измерение. Ниже показан пример объявления зубчатого двумерного массива с самым внешним измерением, составляющим 3:

```
int[][] matrix = new int[3][];
```

Интересно отметить применение конструкции `new int[3][]`, а не `new int[][3]`. Эрик Липперт написал великолепную статью, в которой объясняются причины: `http://albahari.com/jagged`.

Внутренние измерения в объявлении не указываются, т.к. в отличие от прямоугольного массива каждый внутренний массив может иметь произвольную длину. Каждый внутренний массив неявно инициализируется значением `null`, а не пустым массивом. Каждый внутренний массив должен создаваться вручную:

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int[3];  // Создать внутренний массив
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

Зубчатый массив можно инициализировать следующим образом (с целью создания массива, идентичного предыдущему примеру, но с дополнительным элементом в конце):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

## Упрощенные выражения инициализации массивов

Существуют два способа сократить выражения инициализации массивов. Первый из них — опустить операцию new и квалификаторы типов:

```
char[] vowels = {'a','e','i','o','u'};

int[,] rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

int[][] jaggedMatrix =
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Второй подход предусматривает использование ключевого слова var, которое сообщает компилятору о необходимости неявной типизации локальной переменной:

```
var i = 3;            // i неявно получает тип int
var s = "sausage";    // s неявно получает тип string

// Следовательно:

var rectMatrix = new int[,]  // rectMatrix неявно получает тип int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

var jaggedMat = new int[][]  // jaggedMat неявно получает тип int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Неявную типизацию для массивов можно продвинуть на шаг дальше: опустить квалификатор типа после ключевого слова new и позволить компилятору самостоятельно *вывести* тип массива:

```
var vowels = new[] {'a','e','i','o','u'};  // Компилятор выводит тип char[]
```

Чтобы это работало, элементы должны быть неявно преобразуемыми в единственный тип (вдобавок хотя бы один элемент должен относиться к этому типу и должен быть в точности один наилучший тип). Например:

```
var x = new[] {1,10000000000};  // Все элементы преобразуемы в тип long
```

## Проверка границ

Во время выполнения все обращения к индексам массивов проверяются на предмет выхода за допустимые пределы. В случае указания недопустимого значения индекса генерируется исключение IndexOutOfRangeException:

```
int[] arr = new int[3];
arr[3] = 1;  // Генерируется исключение IndexOutOfRangeException
```

Как и в языке Java, проверка границ необходима для обеспечения безопасности типов и упрощения отладки.

Обычно влияние на производительность проверки границ оказывается незначительным, и компилятор JIT способен проводить оптимизацию, такую как заблаговременное выяснение, будут ли все индексы безопасными, перед входом в цикл, устраняя необходимость в выполнении проверки на каждой итерации. Кроме того в языке C# поддерживается "небезопасный" код, который может явно пропускать проверку границ (см. раздел "Небезопасный код и указатели" в главе 4).

# Переменные и параметры

Переменная представляет ячейку в памяти, которая содержит изменяемое значение. Переменная может быть *локальной переменной*, *параметром* (`value`, `ref` либо `out`), *полем* (*экземпляра* либо *статическим*) или *элементом массива*.

## Стек и куча

*Стек* и *куча* – это места, где располагаются переменные и константы. Стек и куча имеют существенно отличающуюся семантику времени жизни.

### Стек

Стек представляет собой блок памяти для хранения локальных переменных и параметров. Стек логически расширяется при входе в функцию и сужается после выхода из нее. Взгляните на следующий метод (чтобы не отвлекать внимание, проверка входного аргумента не делается):

```
static int Factorial (int x)
{
  if (x == 0) return 1;
  return x * Factorial (x-1);
}
```

Метод `Factorial` является рекурсивным, т.е. вызывает сам себя. Каждый раз, когда происходит вход в метод, в стеке размещается экземпляр `int`, а каждый раз, когда метод завершается, экземпляр `int` освобождается.

### Куча

Куча представляет собой блок памяти, в котором располагаются *объекты* (т.е. экземпляры ссылочного типа). Всякий раз, когда создается новый объект, он размещается в куче с возвращением ссылки на созданный объект. Во время выполнения программы куча начинает заполняться по мере создания новых объектов. В исполняющей среде предусмотрен сборщик мусора, который периодически освобождает объекты из кучи, поэтому программа не столкнется с ситуацией нехватки памяти. Объект становится пригодным для освобождения, если на него не ссылается что-то, что само существует.

В приведенном далее примере мы начинаем с создания объекта `StringBuilder`, на который ссылается переменная `ref1`, и выводим на экран его содержимое. Данный объект `StringBuilder` затем немедленно может быть обработан сборщиком мусора, поскольку впоследствии он нигде не задействован.

Затем мы создаем еще один объект `StringBuilder`, на который ссылается переменная `ref2`, и копируем эту ссылку в `ref3`. Хотя `ref2` в дальнейшем не применяется, переменная `ref3` поддерживает существование объекта `StringBuilder`, гарантируя тем самым, что он не будет подвергаться сборке мусора до тех пор, пока не закончится работа с `ref3`.

```
using System;
using System.Text;

class Test
{
  static void Main()
  {
    StringBuilder ref1 = new StringBuilder ("object1");
    Console.WriteLine (ref1);  // object1
    // Объект StringBuilder, на который ссылается ref1,
    // теперь пригоден для сборки мусора

    StringBuilder ref2 = new StringBuilder ("object2");
    StringBuilder ref3 = ref2;
    // Объект StringBuilder, на который ссылается ref2,
    // пока еще НЕ пригоден для сборки мусора

    Console.WriteLine (ref3);  // object2
  }
}
```

Экземпляры типов значений (и ссылки на объекты) хранятся там, где были объявлены соответствующие переменные. Если экземпляр был объявлен как поле внутри типа класса или как элемент массива, то такой экземпляр расположится в куче.

В языке C# нельзя явно удалять объекты, как это можно делать в C++. Объект без ссылок со временем уничтожается сборщиком мусора.

В куче также хранятся статические поля. В отличие от объектов, размещенных в куче (которые могут быть обработаны сборщиком мусора), они существуют до тех пор, пока не будет разрушен домен приложения.

## Определенное присваивание

В C# принудительно применяется политика определенного присваивания. На практике это означает, что за пределами контекста `unsafe` получать доступ к неинициализированной памяти невозможно. Определенное присваивание приводит к трем последствиям.

- Локальным переменным должны быть присвоены значения, прежде чем их можно будет читать.
- При вызове метода должны быть предоставлены аргументы функции (если только они не помечены как необязательные – см. раздел "Необязательные параметры" далее в главе).
- Все остальные переменные (такие как поля и элементы массивов) автоматически инициализируются исполняющей средой.

Например, следующий код приводит к ошибке на этапе компиляции:

```
static void Main()
{
  int x;
  Console.WriteLine (x);  // Ошибка на этапе компиляции
}
```

Поля и элементы массива автоматически инициализируются стандартными значениями для своих типов. Показанный ниже код выводит на экран `0`, потому что элементам массива неявно присвоены их стандартные значения:

```
static void Main()
{
  int[] ints = new int[2];
  Console.WriteLine (ints[0]);  // 0
}
```

Следующий код выводит `0`, т.к. полю неявно присвоено стандартное значение:

```
class Test
{
  static int x;
  static void Main() { Console.WriteLine (x); }  // 0
}
```

## Стандартные значения

Экземпляры всех типов имеют стандартные значения. Стандартные значения для предопределенных типов являются результатом побитового обнуления памяти.

| Тип | Стандартное значение |
|---|---|
| Все ссылочные типы | `null` |
| Все числовые и перечислимые типы | `0` |
| Тип `char` | `'\0'` |
| Тип `bool` | `false` |

Получить стандартное значение для любого типа можно с помощью ключевого слова `default` (на практике оно используется при работе с обобщениями, которые рассматриваются в главе 3):

```
decimal d = default (decimal);
```

Стандартное значение в специальном типе значения (т.е. `struct`) представляет собой то же самое, что и стандартные значения для всех полей, определенных специальным типом.

## Параметры

Метод принимает последовательность параметров. Параметры определяют набор аргументов, которые должны быть предоставлены данному методу. В следующем примере метод `Foo` имеет единственный параметр по имени `p` типа `int`:

```
static void Foo (int p)
{
  p = p + 1;                    // Увеличить p на 1
  Console.WriteLine (p);        // Вывести значение p на экран
}
static void Main()
{
  Foo (8);  // Вызвать Foo с аргументом 8
}
```

Управлять способом передачи параметров можно посредством модификаторов `ref` и `out`.

| Модификатор параметра | Способ передачи | Когда переменная должна быть определенно присвоена |
|---|---|---|
| Отсутствует | По значению | При *входе* |
| `ref` | По ссылке | При *входе* |
| `out` | По ссылке | При *выходе* |

### Передача аргументов по значению

По умолчанию аргументы в C# *передаются по значению*, что общепризнанно является самым распространенным случаем. Другими словами, при передаче значения методу создается его копия:

```
class Test
{
  static void Foo (int p)
  {
    p = p + 1;                  // Увеличить p на 1
    Console.WriteLine (p);      // Вывести значение p на экран
  }

  static void Main()
  {
    int x = 8;
    Foo (x);                    // Создается копия x
    Console.WriteLine (x);      // x по-прежнему будет иметь значение 8
  }
}
```

Присваивание p нового значения не изменяет содержимое x, поскольку p и x находятся в разных ячейках памяти.

Передача по значению аргумента ссылочного типа приводит к копированию *ссылки*, но не объекта. В следующем примере метод Foo видит тот же объект StringBuilder, который был создан в Main, однако имеет независимую *ссылку* на него. Другими словами, sb и fooSB являются отдельными друг от друга переменными, которые ссылаются на один и тот же объект StringBuilder:

```
class Test
{
  static void Foo (StringBuilder fooSB)
  {
    fooSB.Append ("test");
    fooSB = null;
  }

  static void Main()
  {
    StringBuilder sb = new StringBuilder();
    Foo (sb);
    Console.WriteLine (sb.ToString());  // test
  }
}
```

Из-за того, что fooSB – *копия* ссылки, установка ее в null не приводит к установке в null переменной sb. (Тем не менее, если параметр fooSB объявить и вызвать с модификатором ref, то sb *станет* равным null.)

### Модификатор ref

Для *передачи по ссылке* в C# предусмотрен модификатор параметра ref. В приведенном далее примере p и x ссылаются на одну и ту же ячейку памяти:

```
class Test
{
  static void Foo (ref int p)
  {
    p = p + 1;                  // Увеличить p на 1
    Console.WriteLine (p);      // Вывести значение p на экран
  }

  static void Main()
  {
    int x = 8;
    Foo (ref  x);               // Позволить Foo работать напрямую с x
    Console.WriteLine (x);      // x теперь имеет значение 9
  }
}
```

Теперь присваивание p нового значения изменяет содержимое x. Обратите внимание, что модификатор ref должен быть указан как при определении, так и при вызове метода[4]. Подобное требование крайне проясняет происходящее.

[4] Исключением из этого правила является вызов методов COM. Мы обсудим данную тему в главе 25.

Модификатор `ref` критически важен при реализации метода обмена (в разделе "Обобщения" главы 3 мы покажем, как реализовать метод обмена, работающий с любым типом):

```
class Test
{
  static void Swap (ref string a, ref string b)
  {
    string temp = a;
    a = b;
    b = temp;
  }

  static void Main()
  {
    string x = "Penn";
    string y = "Teller";
    Swap (ref x, ref y);
    Console.WriteLine (x);  // Teller
    Console.WriteLine (y);  // Penn
  }
}
```

Параметр может быть передан по ссылке или по значению вне зависимости от того, относится он к ссылочному типу или к типу значения.

### Модификатор `out`

Аргумент `out` похож на аргумент `ref` за исключением следующих аспектов:

- он не нуждается в присваивании значения перед входом в функцию;
- ему должно быть присвоено значение перед выходом из функции.

Модификатор `out` чаще всего применяется для получения из метода нескольких возвращаемых значений. Например:

```
class Test
{
  static void Split (string name, out string firstNames,
                     out string lastName)
  {
    int i = name.LastIndexOf (' ');
    firstNames = name.Substring (0, i);
    lastName   = name.Substring (i + 1);
  }

  static void Main()
  {
    string a, b;
    Split ("Stevie Ray Vaughan", out a, out b);
    Console.WriteLine (a);  // Stevie Ray
    Console.WriteLine (b);  // Vaughan
  }
}
```

Подобно параметру `ref` параметр `out` передается по ссылке.

### Переменные out и отбрасывание (C# 7)

В версии C# 7 переменные можно объявлять на лету при вызове методов с параметрами out. Вот как сократить метод Main из предыдущего примера:

```
static void Main()
{
  Split ("Stevie Ray Vaughan", out string a, out string b);
  Console.WriteLine (a);  // Stevie Ray
  Console.WriteLine (b);  // Vaughan
}
```

Иногда при вызове методов с многочисленными параметрами out вы не заинтересованы в получении значений из всех параметров. В таких ситуациях вы можете с помощью символа подчеркивания "отбросить" те параметры, которые не представляют интерес:

```
Split ("Stevie Ray Vaughan", out string a, out _);     // Отбросить второй
                                                       // параметр out
Console.WriteLine (a);
```

В данном случае компилятор трактует символ подчеркивания как специальный символ, называемый *отбрасыванием*. В одиночный вызов допускается включать множество отбрасываний. Предполагая, что метод SomeBigMethod был определен с семью параметрами out, вот как проигнорировать все кроме четвертого:

```
SomeBigMethod (out _, out _, out _, out int x, out _, out _, out _);
```

В целях обратной совместимости данное языковое средство не вступит в силу, если в области видимости находится реальная переменная с именем в виде символа подчеркивания:

```
string _;
Split ("Stevie Ray Vaughan", out string a, _);  // Не скомпилируется
```

### Последствия передачи по ссылке

Когда вы передаете аргумент по ссылке, то устанавливаете псевдоним для ячейки памяти, в которой находится существующая переменная, а не создаете новую ячейку. В следующем примере переменные x и y представляют один и тот же экземпляр:

```
class Test
{
  static int x;

  static void Main() { Foo (out x); }

  static void Foo (out int y)
  {
    Console.WriteLine (x);     // x имеет значение 0
    y = 1;                     // Изменить значение y
    Console.WriteLine (x);     // x имеет значение 1
  }
}
```

### Модификатор params

Для последнего параметра метода может быть указан модификатор params, позволяя методу принимать любое количество аргументов определенного типа. Тип параметра должен быть объявлен как массив. Например:

```
class Test
{
  static int Sum (params int[] ints)
  {
    int sum = 0;
    for (int i = 0; i < ints.Length; i++)
      sum += ints[i];                // Увеличить sum на ints[i]
    return sum;
  }
  static void Main()
  {
    int total = Sum (1, 2, 3, 4);
    Console.WriteLine (total);       // 10
  }
}
```

Аргумент `params` можно также предоставить как обычный массив. Первая строка кода в `Main` семантически эквивалентна следующей строке:

```
int total = Sum (new int[] { 1, 2, 3, 4 } );
```

### Необязательные параметры

Начиная с версии C# 4.0, в методах, конструкторах и индексаторах (глава 3) можно объявлять *необязательные параметры*. Параметр является необязательным, если в его объявлении указано стандартное значение:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

При вызове метода необязательные параметры могут быть опущены:

```
Foo();  // 23
```

Необязательному параметру `x` в действительности *передается стандартный аргумент* со значением `23` — компилятор встраивает это значение в скомпилированный код на *вызывающей* стороне. Показанный выше вызов `Foo` семантически эквивалентен следующему вызову:

```
Foo (23);
```

поскольку компилятор просто подставляет стандартное значение необязательного параметра там, где он используется.

Добавление необязательного параметра к открытому методу, который вызывается из другой сборки, требует перекомпиляции обеих сборок — как и в случае, если бы параметр был обязательным.

Стандартное значение необязательного параметра должно быть указано в виде константного выражения или вызова конструктора без параметров для типа значения. Необязательные параметры не могут быть помечены как `ref` или `out`.

Обязательные параметры должны находиться *перед* необязательными параметрами в объявлении метода и его вызове (исключением являются аргументы `params`, которые всегда располагаются в конце). В следующем примере параметру `x` передается явное значение `1`, а параметру `y` — стандартное значение `0`:

```
void Foo (int x = 0, int y = 0) { Console.WriteLine (x + ", " + y); }

void Test()
{
  Foo(1);  // 1, 0
}
```

Чтобы сделать обратное (передать стандартное значение для x и явное значение для y), потребуется скомбинировать необязательные параметры с *именованными аргументами*.

### Именованные аргументы

Вместо распознавания аргумента по позиции его можно идентифицировать по имени. Например:

```
void Foo (int x, int y) { Console.WriteLine (x + ", " + y); }

void Test()
{
  Foo (x:1, y:2);   // 1, 2
}
```

Именованные аргументы могут указываться в любом порядке. Следующие вызовы Foo семантически идентичны:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

Тонкое отличие состоит в том, что выражения в аргументах вычисляются согласно порядку, в котором они появляются на *вызывающей* стороне. В общем случае это актуально только для взаимозависимых выражений с побочными эффектами, как в следующем коде, выводящем на экран 0, 1:

```
int a = 0;
Foo (y: ++a, x: --a);  // Выражение ++a вычисляется первым
```

Разумеется, на практике вы определенно должны избегать подобного стиля кодирования!

Именованные и позиционные аргументы можно смешивать:

```
Foo (1, y:2);
```

Однако существует одно ограничение: позиционные аргументы должны находиться перед именованными аргументами. Таким образом, вызвать Foo, как показано ниже, не удастся:

```
Foo (x:1, 2);  // Ошибка на этапе компиляции
```

Именованные аргументы особенно удобны в сочетании с необязательными параметрами. Например, взгляните на следующий метод:

```
void Bar (int a = 0, int b = 0, int c = 0, int d = 0) { ... }
```

Его можно вызвать, предоставив только значение для d:

```
Bar (d:3);
```

Это очень удобно при работе с API-интерфейсами COM, как будет обсуждаться в разделе "Собственная возможность взаимодействия и возможность взаимодействия с COM" главы 5.

## Ссылочные локальные переменные (C# 7)

В версии C# 7 появилось экзотическое средство, позволяющее определить локальную переменную, которая *ссылается* на элемент в массиве или на поле в объекте:

```
int[] numbers = { 0, 1, 2, 3, 4 };
ref int numRef = ref numbers [2];
```

В данном примере numRef — *ссылка* на numbers[2]. Модификация numRef приводит к модификации элемента массива:

```
numRef *= 10;
Console.WriteLine (numRef);        // 20
Console.WriteLine (numbers [2]);   // 20
```

В качестве цели ссылочной локальной переменной должен указываться элемент массива, поле или обычная локальная переменная; ею не может быть свойство (глава 3). *Ссылочные локальные переменные* предназначены для специализированных сценариев микро-оптимизации и обычно применяются в сочетании с *возвращаемыми ссылочными значениями*.

## Возвращаемые ссылочные значения (C# 7)

Ссылочную локальную переменную можно возвращать из метода. Результат называется *возвращаемым ссылочным значением*:

```
static string X = "Old Value";
static ref string GetX() => ref X;  // Этот метод возвращает ссылочное значение
static void Main()
{
  ref string xRef = ref GetX();     // Присвоить результат ссылочной
                                    // локальной переменной
  xRef = "New Value";
  Console.WriteLine (X);            // Выводит New Value
}
```

## Объявление неявно типизированных локальных переменных с помощью var

Часто случается так, что переменная объявляется и инициализируется за один шаг. Если компилятор способен вывести тип из инициализирующего выражения, то на месте объявления типа можно использовать ключевое слово var (появившееся в версии C# 3.0). Например:

```
var x = "hello";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

Данный код в точности эквивалентен следующему коду:

```
string x = "hello";
System.Text.StringBuilder y = new System.Text.StringBuilder();
float z = (float)Math.PI;
```

По причине такой прямой эквивалентности неявно типизированные переменные являются статически типизированными. Скажем, приведенный ниже код вызовет ошибку на этапе компиляции:

```
var x = 5;
x = "hello";  // Ошибка на этапе компиляции; x относится к типу int
```

Применение var может ухудшить читабельность кода в случае, если *вы не можете вывести тип, просто взглянув на объявление переменной*. Например:

```
Random r = new Random();
var x = r.Next();
```

Какой тип имеет переменная x?

В разделе "Анонимные типы" главы 4 мы опишем сценарий, когда использование ключевого слова `var` обязательно.

# Выражения и операции

*Выражение* по существу обозначает значение. Простейшими видами выражений являются константы и переменные. Выражения могут видоизменяться и комбинироваться с применением операций. *Операция* принимает один или более входных *операндов*, чтобы образовать новое выражение.

Ниже показан пример *константного выражения*:

```
12
```

Посредством операции `*` можно скомбинировать два операнда (литеральные выражения `12` и `30`):

```
12 * 30
```

Можно строить сложные выражения, поскольку операнд сам по себе может быть выражением, как операнд `(12 * 30)` в следующем примере:

```
1 + (12 * 30)
```

Операции в C# могут быть классифицированы как *унарные*, *бинарные* и *тернарные* в зависимости от количества операндов, с которыми они работают (один, два или три). Бинарные операции всегда используют *инфиксную* форму, когда операция помещается *между* двумя операндами.

## Первичные выражения

Первичные выражения включают выражения, сформированные из операций, которые являются неотъемлемой частью самого языка. Ниже показан пример:

```
Math.Log (1)
```

Выражение состоит из двух первичных выражений. Первое выражение осуществляет поиск члена (посредством операции `.`), а второе — вызов метода (с помощью операции `()`).

## Пустые выражения

Пустое выражение — это выражение, которое не имеет значения. Например:

```
Console.WriteLine (1)
```

Поскольку пустое выражение не имеет значения, оно не может применяться в качестве операнда при построении более сложных выражений:

```
1 + Console.WriteLine (1)  // Ошибка на этапе компиляции
```

## Выражения присваивания

Выражение присваивания использует операцию = для присваивания переменной результата вычисления другого выражения. Например:

```
x = x * 5
```

Выражение присваивания — не пустое выражение. Оно заключает в себе значение, которое было присвоено, и потому может встраиваться в другое выражение.

В следующем примере выражение присваивает 2 переменной x и 10 переменной y:

```
y = 5 * (x = 2)
```

Такой стиль выражения может применяться для инициализации нескольких значений:

```
a = b = c = d = 0
```

*Составные операции присваивания* являются синтаксическим сокращением, которое комбинирует присваивание с другой операцией. Например:

```
x *= 2      // Эквивалентно x = x * 2
x <<= 1     // Эквивалентно x = x << 1
```

(Небольшое исключение из данного правила касается *событий*, которые рассматриваются в главе 4: операции += и -= в них трактуются специальным образом и отображаются на средства доступа add и remove события.)

## Приоритеты и ассоциативность операций

Когда выражение содержит несколько операций, порядок их вычисления определяется *приоритетами* и *ассоциативностью*. Операции с более высокими приоритетами выполняются перед операциями, приоритеты которых ниже. Если операции имеют одинаковые приоритеты, то порядок их выполнения определяется ассоциативностью.

### Приоритеты операций

Приведенное ниже выражение:

```
1 + 2 * 3
```

вычисляется следующим образом, потому что операция * имеет больший приоритет, чем +:

```
1 + (2 * 3)
```

### Левоассоциативные операции

Бинарные операции (кроме операции присваивания, лямбда-операции и операции объединения с null) являются *левоассоциативными*; другими словами, они вычисляются слева направо. Например, выражение:

```
8 / 4 / 2
```

по причине левой ассоциативности вычисляется так:

```
( 8 / 4 ) / 2    // 1
```

Чтобы изменить действительный порядок вычисления, можно расставить скобки:

```
8 / ( 4 / 2 )    // 4
```

### Правоассоциативные операции

Операции присваивания, лямбда-операция, операция объединения с null и условная операция являются *правоассоциативными*; другими словами, они вычисляются справа налево. Правая ассоциативность позволяет успешно компилировать множественное присваивание вроде показанного ниже:

```
x = y = 3;
```

Здесь значение 3 сначала присваивается переменной y, после чего результат этого выражения (3) присваивается переменной x.

## Таблица операций

В табл. 2.3 перечислены операции C# в порядке их приоритетов. Операции в одной и той же категории имеют одинаковые приоритеты. Операции, которые могут быть перегружены пользователем, объясняются в разделе "Перегрузка операций" главы 4.

**Таблица 2.3. Операции C# (с категоризацией в порядке приоритетов)**

| Категория | Символ операции | Название операции | Пример | Возможность перегрузки пользователем |
|---|---|---|---|---|
| Первичные | `.` | Доступ к члену | `x.y` | Нет |
| | `->` (небезопасная) | Указатель на структуру | `x->y` | Нет |
| | `()` | Вызов функции | `x()` | Нет |
| | `[]` | Массив/индекс | `a[x]` | Через индексатор |
| | `++` | Постфиксная форма инкремента | `x++` | Да |
| | `--` | Постфиксная форма декремента | `x--` | Да |
| | `new` | Создание экземпляра | `new Foo()` | Нет |
| | `stackalloc` | Небезопасное выделение памяти в стеке | `stackalloc(10)` | Нет |
| | `typeof` | Получение типа по идентификатору | `typeof(int)` | Нет |
| | `nameof` | Получение имени идентификатора | `nameof(x)` | Нет |
| | `checked` | Включение проверки целочисленного переполнения | `checked(x)` | Нет |
| | `unchecked` | Отключение проверки целочисленного переполнения | `unchecked(x)` | Нет |
| | `default` | Стандартное значение | `default(char)` | Нет |
| | `?.` | `null`-условная | `x?.y` | Нет |
| Унарные | `await` | Ожидание | `await myTask` | Нет |
| | `sizeof` | Получение размера структуры | `sizeof(int)` | Нет |
| | `+` | Положительное значение | `+x` | Да |

| Категория | Символ операции | Название операции | Пример | Возможность перегрузки пользователем |
|---|---|---|---|---|
| | - | Отрицательное значение | `-x` | Да |
| | ! | НЕ | `!x` | Да |
| | ~ | Побитовое дополнение | `~x` | Да |
| | ++ | Префиксная форма инкремента | `++x` | Да |
| | -- | Префиксная форма декремента | `--x` | Да |
| | () | Приведение | `(int)x` | Нет |
| | * (небезопасная) | Значение по адресу | `*x` | Нет |
| | & (небезопасная) | Адрес значения | `&x` | Нет |
| Мультипликативные | * | Умножение | `x * y` | Да |
| | / | Деление | `x / y` | Да |
| | % | Остаток от деления | `x % y` | Да |
| Аддитивные | + | Сложение | `x + y` | Да |
| | - | Вычитание | `x - y` | Да |
| Сдвига | << | Сдвиг влево | `x << 1` | Да |
| | >> | Сдвиг вправо | `x >> 1` | Да |
| Отношения | < | Меньше | `x < y` | Да |
| | > | Больше | `x > y` | Да |
| | <= | Меньше или равно | `x <= y` | Да |
| | >= | Больше или равно | `x >= y` | Да |
| | is | Принадлежность к типу или его подклассу | `x is y` | Нет |
| | as | Преобразование типа | `x as y` | Нет |
| Эквивалентности | == | Равно | `x == y` | Да |
| | != | Не равно | `x != y` | Да |
| Логическое И | & | И | `x & y` | Да |
| Логическое исключающее ИЛИ | ^ | Исключающее ИЛИ | `x ^ y` | Да |

*Окончание табл. 2.3*

| Категория | Символ операции | Название операции | Пример | Возможность перегрузки пользователем |
|---|---|---|---|---|
| Логическое ИЛИ | `\|` | ИЛИ | `x \| y` | Да |
| Условное И | `&&` | Условное И | `x && y` | Через `&` |
| Условное ИЛИ | `\|\|` | Условное ИЛИ | `x \|\| y` | Через `\|` |
| Объединение с null | `??` | Объединение с null | `x ?? y` | Нет |
| Условная | `?:` | Условная | `isTrue ? thenThisValue: elseThisValue` | Нет |
| Присваивание и лямбда | `=` | Присваивание | `x = y` | Нет |
| | `*=` | Умножение с присваиванием | `x *= 2` | Через `*` |
| | `/=` | Деление с присваиванием | `x /= 2` | Через `/` |
| | `+=` | Сложение с присваиванием | `x += 2` | Через `+` |
| | `-=` | Вычитание с присваиванием | `x -= 2` | Через `-` |
| | `<<=` | Сдвиг влево с присваиванием | `x <<= 2` | Через `<<` |
| | `>>=` | Сдвиг вправо с присваиванием | `x >>= 2` | Через `>>` |
| | `&=` | Операция И с присваиванием | `x &= 2` | Через `&` |
| | `^=` | Операция исключающего ИЛИ с присваиванием | `x ^= 2` | Через `^` |
| | `\|=` | Операция ИЛИ с присваиванием | `x \|= 2` | Через `\|` |
| | `=>` | Лямбда-операция | `x => x + 1` | Нет |

# Операции для работы со значениями `null`

В языке C# предоставляются две операции, которые предназначены для упрощения работы со значениями `null`: *операция объединения с `null`* (null coalescing) и *`null`-условная операция* (null-conditional).

## Операция объединения с null

*Операция объединения с null* обозначается как ??. Она выполняется следующим образом: если операнд не равен null, тогда возвращается его значение, а иначе возвращается стандартное значение. Например:

```
string s1 = null;
string s2 = s1 ?? "nothing";  // Переменная s2 получает значение "nothing"
```

Если левостороннее выражение не равно null, то правостороннее выражение никогда не вычисляется. Операция объединения с null также работает с типами, допускающими null (см. раздел "Типы, допускающие значение null" в главе 4).

## null-условная операция (C# 6)

*null-условная операция* (или *элвис-операция*) обозначается как ?. и является нововведением версии C# 6. Она позволяет вызывать методы и получать доступ к членам подобно стандартной операции точки, но с той разницей, что если находящийся слева операнд равен null, то результатом выражения будет null, а не генерация исключения NullReferenceException:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();     // Ошибка не возникает;
                               // взамен s получает значение null
```

Последняя строка кода эквивалентна следующему коду:

```
string s = (sb == null ? null : sb.ToString());
```

Встретив значение null, элвис-операция сокращает вычисление оставшейся части выражения. В приведенном далее примере переменная s получает значение null, несмотря на наличие стандартной операции точки между ToString и ToUpper:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString().ToUpper();   // Переменная s получает значение
                                       // null и ошибка не возникает
```

Многократное использование элвис-операции необходимо, только если находящийся непосредственно слева операнд может быть равен null. Следующее выражение надежно работает в ситуациях, когда и x, и x.y могут быть равны null:

```
x?.y?.z
```

Оно эквивалентно такому выражению (за исключением того, что x.y оценивается только один раз):

```
x == null ? null
        : (x.y == null ? null : x.y.z)
```

Окончательное выражение должно быть способным принимать значение null. Показанный ниже код не является допустимым, потому что переменная типа int не может принимать значение null:

```
System.Text.StringBuilder sb = null;
int length = sb?.ToString().Length;    // Не допускается: переменная int
                                       // не может принимать значение null
```

Исправить положение можно за счет применения типа значения, допускающего null (см. раздел "Типы, допускающие значение null" в главе 4). На тот случай, если вы уже знакомы с типами, допускающими null, то вот как выглядит код:

```
int? length = sb?.ToString().Length;  // Допустимо: переменная int?
                                      // может принимать значение null
```

null-условную операцию можно также использовать для вызова метода void:

```
someObject?.SomeVoidMethod();
```

Если переменная someObject равна null, тогда такой вызов становится "отсутствием операции" вместо того, чтобы приводить к генерации исключения NullReferenceException.

null-условная операция может применяться с часто используемыми членами типов, которые будут описаны в главе 3, в том числе с *методами*, *полями*, *свойствами* и *индексаторами*. Она также хорошо сочетается с операцией объединения с null:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString() ?? "nothing"; //Переменная s получает значение "nothing"
```

# Операторы

Функции состоят из операторов, которые выполняются последовательно в порядке их появления внутри программы. *Блок операторов* — это последовательность операторов, находящихся между фигурными скобками ({}).

## Операторы объявления

Оператор объявления объявляет новую переменную и может дополнительно инициализировать ее посредством выражения. Оператор объявления завершается точкой с запятой. Можно объявлять несколько переменных одного и того же типа, указывая их в списке с запятой в качестве разделителя. Например:

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

Объявление константы похоже на объявление переменной за исключением того, что после объявления константа не может быть изменена и объявление обязательно должно сопровождаться инициализацией (см. раздел "Константы" в главе 3):

```
const double c = 2.99792458E08;
c += 10;  // Ошибка на этапе компиляции
```

### Локальные переменные

Областью видимости локальной переменной или локальной константы является текущий блок. Объявлять еще одну локальную переменную с тем же самым именем в текущем блоке или в любых вложенных блоках не разрешено. Например:

```
static void Main()
{
  int x;
  {
    int y;
    int x;  // Ошибка - переменная x уже определена
  }
  {
    int y;  // Нормально - переменная y не находится в области видимости
  }
  Console.Write (y); // Ошибка - переменная y находится
                     // за пределами области видимости
}
```

Область видимости переменной распространяется в обоих направлениях на всем протяжении ее блока кода. Это означает, что даже если в приведенном примере перенести первоначальное объявление x в конец метода, то будет получена та же ошибка. Поведение отличается от языка C++ и в чем-то необычно, учитывая недопустимость ссылки на переменную или константу до ее объявления.

## Операторы выражений

Операторы выражений представляют собой выражения, которые также являются допустимыми операторами. Оператор выражения должен либо изменять состояние, либо вызывать что-то, что может изменять состояние. Изменение состояния по существу означает изменение переменной. Ниже перечислены возможные операторы выражений:

- выражения присваивания (включая выражения инкремента и декремента);
- выражения вызова методов (`void` и не `void`);
- выражения создания экземпляров объектов.

Рассмотрим несколько примеров:

```
// Объявить переменные с помощью операторов объявления:
string s;
int x, y;
System.Text.StringBuilder sb;

// Операторы выражений
x = 1 + 2;                    // Выражение присваивания
x++;                          // Выражение инкремента
y = Math.Max (x, 5);          // Выражение присваивания
Console.WriteLine (y);        // Выражение вызова метода
sb = new StringBuilder();     // Выражение присваивания
new StringBuilder();          // Выражение создания экземпляра объекта
```

При вызове конструктора или метода, который возвращает значение, вы не обязаны использовать результат. Тем не менее, если этот конструктор или метод не изменяет состояние, то такой оператор совершенно бесполезен:

```
new StringBuilder();          // Допустим, но бесполезен
new string ('c', 3);          // Допустим, но бесполезен
x.Equals (y);                 // Допустим, но бесполезен
```

## Операторы выбора

В C# имеются следующие механизмы для условного управления потоком выполнения программы:

- операторы выбора (`if`, `switch`);
- условная операция (`?:`);
- операторы цикла (`while`, `do..while`, `for`, `foreach`).

В текущем разделе рассматриваются две простейших конструкции: оператор `if-else` и оператор `switch`.

### Оператор `if`

Оператор `if` выполняет некоторый оператор, если выражение `bool` в результате дает `true`. Например:

```
if (5 < 2 * 3)
  Console.WriteLine ("true");  // Выводит true
```

В качестве оператора может выступать блок кода:

```
if (5 < 2 * 3)
{
  Console.WriteLine ("true");
  Console.WriteLine ("Let's move on!");
}
```

### Конструкция `else`

Оператор `if` может быть дополнительно снабжен конструкцией `else`:

```
if (2 + 2 == 5)
  Console.WriteLine ("Does not compute");
else
  Console.WriteLine ("False");  // Выводит False
```

Внутрь конструкции `else` можно помещать другой оператор `if`:

```
if (2 + 2 == 5)
  Console.WriteLine ("Does not compute");
else
  if (2 + 2 == 4)
    Console.WriteLine ("Computes");  // Выводит Computes
```

### Изменение потока выполнения с помощью фигурных скобок

Конструкция `else` всегда применяется к непосредственно предшествующему оператору `if` в блоке операторов. Например:

```
if (true)
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");  // выполняется
```

Код семантически идентичен следующему коду:

```
if (true)
{
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");  // выполняется
}
```

Переместив фигурные скобки, поток выполнения можно изменить:

```
if (true)
{
  if (false)
    Console.WriteLine();
}
else
  Console.WriteLine ("does not execute");  // не выполняется
```

С помощью фигурных скобок вы явно заявляете о своих намерениях. Фигурные скобки могут улучшить читабельность вложенных операторов `if`, даже когда они не требуются компилятором. Важным исключением является следующий шаблон:

```
static void TellMeWhatICanDo (int age)
{
  if (age >= 35)
    Console.WriteLine ("You can be president!"); // Вы можете стать президентом!
  else if (age >= 21)
    Console.WriteLine ("You can drink!");       // Вы можете выпивать!
  else if (age >= 18)
    Console.WriteLine ("You can vote!");        // Вы можете голосовать!
  else
    Console.WriteLine ("You can wait!");        // Вы можете лишь ждать!
}
```

Здесь мы организовали операторы `if` и `else` так, чтобы сымитировать конструкцию "elseif" из других языков (и директиву препроцессора `#elif` в C#). Средство автоформатирования Visual Studio распознает такой шаблон и сохраняет отступы. Однако семантически каждый оператор `if`, следующий за `else`, функционально вложен внутрь конструкции `else`.

### Оператор `switch`

Операторы `switch` позволяют организовать ветвление потока выполнения программы на основе выбора из возможных значений, которые переменная может принимать. Операторы `switch` могут дать в результате более ясный код, чем множество операторов `if`, поскольку они требуют только однократной оценки выражения. Например:

```
static void ShowCard(int cardNumber)
{
  switch (cardNumber)
  {
    case 13:
      Console.WriteLine ("King");       // Король
      break;
    case 12:
      Console.WriteLine ("Queen");      // Дама
      break;
    case 11:
      Console.WriteLine ("Jack");       // Валет
      break;
    case -1:          // Джокер соответствует -1
      goto case 12;   // В этой игре джокер подсчитывается как дама
    default:          // Выполняется для любого другого значения cardNumber
      Console.WriteLine (cardNumber);
      break;
  }
}
```

В приведенном примере демонстрируется самый распространенный сценарий, при котором осуществляется переключение по *константам*. При указании константы вы ограничены встроенными целочисленными типами, `bool`, `char`, `enum` и `string`.

В конце каждой конструкции `case` посредством одного из операторов перехода необходимо явно указывать, куда выполнение должно передаваться дальше (если толь-

ко вы не хотите получить сквозное выполнение). Ниже перечислены возможные варианты:

- `break` (переход в конец оператора `switch`);
- `goto case` *x* (переход на другую конструкцию `case`);
- `goto default` (переход на конструкцию `default`);
- любой другой оператор перехода, а именно — `return`, `throw`, `continue` или `goto` *метка*.

Когда для нескольких значений должен выполняться тот же самый код, конструкции `case` можно записывать последовательно:

```
switch (cardNumber)
{
  case 13:
  case 12:
  case 11:
    Console.WriteLine ("Face card");   // Фигурная карта
    break;
  default:
    Console.WriteLine ("Plain card");  // Нефигурная карта
    break;
}
```

Такая особенность оператора `switch` может иметь решающее значение в плане обеспечения более ясного кода, чем в случае множества операторов `if-else`.

### Оператор `switch` с шаблонами (C# 7)

В версии C# 7 можно переключаться также на основе *типов*:

```
static void Main()
{
  TellMeTheType (12);
  TellMeTheType ("hello");
  TellMeTheType (true);
}

static void TellMeTheType (object x)  // Тип object допускает переменную
                                      // любого типа
{
  switch (x)
  {
    case int i:
      Console.WriteLine ("It's an int!");  // Это не целочисленное значение!
      Console.WriteLine ($"The square of {i} is {i * i}");  // Вывод квадрата
      break;
    case string s:
      Console.WriteLine ("It's a string");  // Это не строка!
      Console.WriteLine ($"The length of {s} is {s.Length}");
                        // Вывод длины строки
      break;
    default:
      Console.WriteLine ("I don't know what x is"); // Неизвестное значение
      break;
  }
}
```

(Тип object разрешает переменную любого типа; мы подробно обсудим это в разделах "Наследование" и "Тип object" главы 3.)

В каждой конструкции case указываются тип для сопоставления и переменная, которой нужно присвоить типизированное значение в случае совпадения (так называемая "шаблонная" переменная). В отличие от констант никаких ограничений на используемые типы не налагается.

Конструкцию case можно снабдить ключевым словом when:

```
switch (x)
{
  case bool b when b == true:  // Выполняется, только если b равно true
    Console.WriteLine ("True!");
    break;
  case bool b:
    Console.WriteLine ("False!");
    break;
}
```

При переключении на основе типов порядок следования конструкций case может быть важным (в отличие от случая с переключением по константам). Рассмотренный пример давал бы другой результат, если бы мы поменяли местами две конструкции case (на самом деле он даже не скомпилировался бы, поскольку компилятор определил бы, что вторая конструкция case недостижима). Исключением из данного правила является конструкция default, которая всегда выполняется последней независимо от того, где находится.

Можно указывать друг за другом несколько конструкций case. Вызов Console.WriteLine в показанном ниже коде будет выполняться для любого значения с плавающей точкой, которое больше 1000:

```
switch (x)
{
  case float f when f > 1000:
  case double d when d > 1000:
  case decimal m when m > 1000:
    Console.WriteLine ("We can refer to x here but not f or d or m");
                    // Мы можем здесь ссылаться на x, но не на f или d или m
    break;
}
```

В приведенном примере компилятор разрешает употреблять шаблонные переменные f, d и m только в конструкциях when. При вызове Console.WriteLine неизвестно, какая из трех переменных будет присвоена, а потому компилятор выносит всех их за пределы области видимости.

В рамках одного оператора switch константы и шаблоны можно по-разному смешивать и сочетать. Вдобавок можно переключаться по значению null:

```
case null:
  Console.WriteLine ("Nothing here");
  break;
```

## Операторы итераций

Язык C# позволяет многократно выполнять последовательность операторов с помощью операторов while, do-while, for и foreach.

### Циклы while и do-while

Циклы while многократно выполняют код в своем теле до тех пор, пока результатом выражения типа bool является true. Выражение проверяется *перед* выполнением тела цикла. Например:

```
int i = 0;
while (i < 3)
{
  Console.WriteLine (i);
  i++;
}
```

```
ВЫВОД:
0
1
2
```

Циклы do-while отличаются по функциональности от циклов while только тем, что выражение в них проверяется *после* выполнения блока операторов (гарантируя, что блок выполнится минимум один раз). Ниже приведен предыдущий пример, переписанный с целью применения цикла do-while:

```
int i = 0;
do
{
  Console.WriteLine (i);
  i++;
}
while (i < 3);
```

### Циклы for

Циклы for похожи на циклы while, но имеют специальные конструкции для *инициализации* и *итерации* переменной цикла. Цикл for содержит три конструкции:

```
for (конструкция-инициализации; конструкция-условия; конструкция-итерации)
  оператор-или-блок-операторов
```

*Конструкция инициализации*

Выполняется перед началом цикла; служит для инициализации одной или большего количества переменных *итерации*.

*Конструкция условия*

Выражение типа bool, при значении true которого будет выполняться тело цикла.

*Конструкция итерации*

Выполняется *после* каждого прогона блока операторов; обычно используется для обновления переменной итерации.

Например, следующий цикл выводит числа от 0 до 2:

```
for (int i = 0; i < 3; i++)
  Console.WriteLine (i);
```

Приведенный ниже код выводит первые 10 чисел последовательности Фибоначчи (где каждое число является суммой двух предыдущих):

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
  Console.WriteLine (prevFib);
  int newFib = prevFib + curFib;
  prevFib = curFib; curFib = newFib;
}
```

Любую из трех частей оператора `for` разрешено опускать. Можно реализовать бесконечный цикл вроде показанного далее (хотя взамен подойдет и `while(true)`):

```
for (;;)
  Console.WriteLine ("interrupt me");
```

### Циклы `foreach`

Оператор `foreach` обеспечивает проход по всем элементам в перечислимом объекте. Большинство типов в C# и .NET Framework, которые представляют набор или список элементов, являются перечислимыми. Примерами перечислимых типов могут служить массивы и строки. Ниже приведен код для перечисления символов в строке, от первого до последнего:

```
foreach (char c in "beer")  // c - переменная итерации
  Console.WriteLine (c);

ВЫВОД:
b
e
e
r
```

Перечислимые объекты описаны в разделе "Перечисление и итераторы" главы 4.

## Операторы перехода

Операторами перехода в C# являются `break`, `continue`, `goto`, `return` и `throw`.

Операторы перехода подчиняются правилам надежности операторов `try` (см. раздел "Операторы `try` и исключения" в главе 4). Это означает следующее:

- при переходе из блока `try` всегда выполняется блок `finally` оператора `try` перед достижением цели перехода;
- переход не может производиться изнутри блока `finally` наружу (исключая применение оператора `throw`).

### Оператор `break`

Оператор `break` завершает выполнение тела итерации или оператора `switch`:

```
int x = 0;
while (true)
{
  if (x++ > 5)
    break;  // Прервать цикл
}
// После break выполнение продолжится здесь
...
```

### Оператор continue

Оператор continue пропускает оставшиеся операторы в цикле и начинает следующую итерацию. В представленном ниже цикле пропускаются четные числа:

```
for (int i = 0; i < 10; i++)
{
  if ((i % 2) == 0)   // Если значение i четное,
    continue;         // перейти к следующей итерации

  Console.Write (i + " ");
}
ВЫВОД: 1 3 5 7 9
```

### Оператор goto

Оператор goto переносит выполнение на указанную метку внутри блока операторов. Он имеет следующую форму:

```
goto метка-оператора;
```

или же такую форму, когда используется внутри оператора switch:

```
goto case константа-case; // Работает только с константами, но не с шаблонами!
```

Метка — это заполнитель в блоке кода, который предваряет оператор и завершается двоеточием. Следующий код выполняет итерацию по числам от 1 до 5, имитируя поведение цикла for:

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write (i + " ");
  i++;
  goto startLoop;
}
ВЫВОД: 1 2 3 4 5
```

Форма goto case *константа-case* переносит поток выполнения на другую конструкцию case в блоке switch (см. раздел "Оператор switch" ранее в главе).

### Оператор return

Оператор return завершает метод и должен возвращать выражение возвращаемого типа метода, если метод не является void:

```
static decimal AsPercentage (decimal d)
{
  decimal p = d * 100m;
  return p;  // Возвратиться в вызывающий метод со значением
}
```

Оператор return может находиться в любом месте метода (кроме блока finally).

### Оператор throw

Оператор throw генерирует исключение, чтобы указать на возникновение ошибки (см. раздел "Операторы try и исключения" в главе 4):

```
if (w == null)
  throw new ArgumentNullException (...);
```

## Смешанные операторы

Оператор `using` предоставляет элегантный синтаксис для вызова метода `Dispose` на объектах, которые реализуют интерфейс `IDisposable`, внутри блока `finally` (см. раздел "Операторы `try` и исключения" в главе 4 и раздел "`IDisposable`, `Dispose` и `Close`" в главе 12).

В C# ключевое слово `using` перегружено, поэтому в разных контекстах оно имеет разный смысл. В частности, *директива* `using` отличается от *оператора* `using`.

Оператор `lock` является сокращением для вызова методов `Enter` и `Exit` класса `Monitor` (главы 14 и 23).

# Пространства имен

Пространство имен – это область, предназначенная для имен типов. Типы обычно организуются в иерархические пространства имен, упрощая их поиск и устраняя возможность возникновения конфликтов. Например, тип `RSA`, который поддерживает шифрование открытым ключом, определен в следующем пространстве имен:

```
System.Security.Cryptography
```

Пространство имен – неотъемлемая часть имени типа. В приведенном далее коде производится вызов метода `Create` класса `RSA`:

```
System.Security.Cryptography.RSA rsa =
  System.Security.Cryptography.RSA.Create();
```

Пространства имен не зависят от сборок, которые являются единицами развертывания, такими как `.exe` или `.dll` (глава 18).

Пространства имен также не влияют на видимость членов – `public`, `internal`, `private` и т.д.

Ключевое слово `namespace` определяет пространство имен для типов внутри данного блока. Например:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
  class Class2 {}
}
```

С помощью точек отражается иерархия вложенных пространств имен. Следующий код семантически идентичен предыдущему примеру:

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

Ссылаться на тип можно с помощью его *полностью заданного имени*, которое включает все пространства имен, от самого внешнего до самого внутреннего. Например, мы могли бы сослаться на `Class1` из предшествующего примера в форме `Outer.Middle.Inner.Class1`.

Говорят, что типы, которые не определены в каком-либо пространстве имен, находятся в *глобальном пространстве имен*. Глобальное пространство имен также включает пространства имен верхнего уровня, такие как `Outer` в приведенном примере.

## Директива `using`

Директива `using` *импортирует* пространство имен, позволяя ссылаться на типы без указания их полностью заданных имен. В следующем коде импортируется пространство имен `Outer.Middle.Inner` из предыдущего примера:

```
using Outer.Middle.Inner;
class Test
{
  static void Main()
  {
    Class1 c;  // Полностью заданное имя указывать не обязательно
  }
}
```

Вполне законно (и часто желательно) определять в двух разных пространствах имен одно и то же имя типа. Тем не менее, обычно это делается только в случаях, когда ситуация, что пользователь пожелает импортировать сразу оба пространства имен, маловероятна. Хорошим примером из .NET Framework может служить класс `TextBox`, который определен и в `System.Windows.Controls` (WPF), и в `System.Web.UI.WebControls` (ASP.NET).

## Директива `using static` (C# 6)

В версии C# 6 появилась возможность импортировать не только пространство имен, но и отдельный тип с помощью директивы `using static`. Затем все статические члены данного типа можно использовать, не снабжая их именем типа. В показанном ниже примере вызывается статический метод `WriteLine` класса `Console`:

```
using static System.Console;
class Test
{
  static void Main() { WriteLine ("Hello"); }
}
```

Директива `using static` импортирует все доступные статические члены типа, включая поля, свойства и вложенные типы (глава 3). Ее можно также применять к перечислимым типам (см. главу 3), что приведет к импортированию их членов. Таким образом, если мы импортируем следующий перечислимый тип:

```
using static System.Windows.Visibility;
```

то вместо `Visibility.Hidden` сможем указывать просто `Hidden`:

```
var textBox = new TextBox { Visibility = Hidden };  // Стиль XAML
```

Если между несколькими директивами `using static` возникнет неоднозначность, тогда компилятор C# не сможет вывести корректный тип из контекста и сообщит об ошибке.

## Правила внутри пространства имен

### Область видимости имен

Имена, объявленные во внешних пространствах имен, могут использоваться во внутренних пространствах имен без дополнительного указания пространства. В следующем примере `Class1` не нуждается в указании пространства имен внутри `Inner`:

```
namespace Outer
{
  class Class1 {}
  namespace Inner
  {
    class Class2 : Class1 {}
  }
}
```

Если ссылаться на тип необходимо из другой ветви иерархии пространств имен, то можно применять частично заданное имя. В показанном ниже примере класс `SalesReport` основан на `Common.ReportBase`:

```
namespace MyTradingCompany
{
  namespace Common
  {
    class ReportBase {}
  }
  namespace ManagementReporting
  {
    class SalesReport : Common.ReportBase {}
  }
}
```

### Сокрытие имен

Если одно и то же имя типа встречается и во внутреннем, и во внешнем пространстве имен, то преимущество получает тип из внутреннего пространства имен. Чтобы сослаться на тип из внешнего пространства имен, имя потребуется уточнить. Например:

```
namespace Outer
{
  class Foo { }

  namespace Inner
  {
    class Foo { }

    class Test
    {
      Foo f1;              // = Outer.Inner.Foo
      Outer.Foo f2;        // = Outer.Foo
    }
  }
}
```

Все имена типов во время компиляции преобразуются в полностью заданные имена. Код на промежуточном языке (Intermediate Language — IL) не содержит неполных или частично заданных имен.

### Повторяющиеся пространства имен

Объявление пространства имен можно повторять при условии, что имена типов внутри пространств имен не конфликтуют друг с другом:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

Код в этом примере можно даже разнести по двум исходным файлам, что позволит компилировать каждый класс в отдельную сборку.

*Исходный файл 1*:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
```

*Исходный файл 2*:

```
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

### Вложенные директивы `using`

Директиву `using` можно вкладывать внутрь пространства имен. Это позволяет ограничить область видимости директивы `using` объявлением пространства имен. В следующем примере имя `Class1` доступно в одной области видимости, но не доступно в другой:

```
namespace N1
{
  class Class1 {}
}
namespace N2
{
  using N1;
  class Class2 : Class1 {}
}
namespace N2
{
  class Class3 : Class1 {}  // Ошибка на этапе компиляции
}
```

## Назначение псевдонимов типам и пространствам имен

Импортирование пространства имен может привести к конфликту имен типов. Вместо полного пространства имен можно импортировать только конкретные типы, которые нужны, и назначать каждому типу псевдоним. Например:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

Псевдоним можно назначить целому пространству имен:

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

# Дополнительные возможности пространств имен

## Внешние псевдонимы

Внешние псевдонимы позволяют программе ссылаться на два типа с одним и тем же полностью заданным именем (т.е. сочетания пространства имен и имени типа одинаковы). Это необычный сценарий, который может возникнуть только в ситуации, когда два типа поступают из разных сборок. Рассмотрим представленный ниже пример.

*Библиотека 1*:

```
// csc target:library /out:Widgets1.dll widgetsv1.cs
namespace Widgets
{
  public class Widget {}
}
```

*Библиотека 2:*

```
// csc target:library /out:Widgets2.dll widgetsv2.cs
namespace Widgets
{
  public class Widget {}
}
```

*Приложение:*

```
// csc /r:Widgets1.dll /r:Widgets2.dll application.cs
using Widgets;
class Test
{
  static void Main()
  {
    Widget w = new Widget();
  }
}
```

Код такого приложения не может быть скомпилирован, потому что имеется неоднозначность с `Widget`. Решить проблему неоднозначности в приложении помогут внешние псевдонимы:

```
// csc /r:W1=Widgets1.dll /r:W2=Widgets2.dll application.cs
extern alias W1;
extern alias W2;
class Test
{
  static void Main()
  {
    W1.Widgets.Widget w1 = new W1.Widgets.Widget();
    W2.Widgets.Widget w2 = new W2.Widgets.Widget();
  }
}
```

## Квалификаторы псевдонимов пространств имен

Как упоминалось ранее, имена во внутренних пространствах имен скрывают имена из внешних пространств. Тем не менее, иногда даже применение полностью заданного имени не устраняет конфликт. Взгляните на следующий пример:

```
namespace N
{
  class A
  {
    public class B {}                          // Вложенный тип
    static void Main() { new A.B(); }          // Создать экземпляр класса B
  }
}
namespace A
{
  class B {}
}
```

Метод `Main` мог бы создавать экземпляр либо вложенного класса `B`, либо класса `B` из пространства имен `A`. Компилятор всегда назначает более высокий приоритет идентификаторам из текущего пространства имен; в данном случае — вложенному классу `B`. Для разрешения конфликтов подобного рода к названию пространства имен можно добавить квалификатор, имеющий отношение к одному из следующих аспектов:

- глобальное пространство имен — корень всех пространств имен (идентифицируется контекстным ключевым словом `global`);
- набор внешних псевдонимов.

Для указания псевдонима пространства имен используется маркер `::`. В этом примере мы указываем на необходимость применения глобального пространства имен (чаще всего подобное можно наблюдать в автоматически генерируемом коде, что призвано устранять конфликты имен):

```
namespace N
{
  class A
  {
    static void Main()
    {
      System.Console.WriteLine (new A.B());
      System.Console.WriteLine (new global::A.B());
    }
    public class B {}
  }
}
namespace A
{
  class B {}
}
```

Ниже приведен пример указания псевдонима (взятый из примера в разделе "Внешние псевдонимы" ранее в главе):

```
extern alias W1;
extern alias W2;
class Test
{
  static void Main()
  {
    W1::Widgets.Widget w1 = new W1::Widgets.Widget();
    W2::Widgets.Widget w2 = new W2::Widgets.Widget();
  }
}
```

# 3

# Создание типов в C#

В настоящей главе мы займемся исследованием типов и членов типов.

## Классы

Класс является наиболее распространенной разновидностью ссылочного типа. Самое простое из возможных объявление класса выглядит следующим образом:

```
class YourClassName
{
}
```

Более сложный класс может дополнительно иметь перечисленные ниже компоненты.

| | |
|---|---|
| Перед ключевым словом `class` | *Атрибуты* и *модификаторы класса*. Модификаторами невложенных классов являются `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe` и `partial` |
| После `YourClassName` | *Параметры обобщенных типов*, *базовый класс* и *интерфейсы* |
| Внутри фигурных скобок | *Члены класса* (к ним относятся *методы*, *свойства*, *индексаторы*, *события*, *поля*, *конструкторы*, *перегруженные операции*, *вложенные типы* и *финализатор*) |

В текущей главе описаны все конструкции кроме атрибутов, функций операций и ключевого слова `unsafe`, которые рассматриваются в главе 4. В последующих разделах члены класса раскрываются по очереди.

### Поля

*Поле* – это переменная, которая является членом класса или структуры. Например:

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

С полями разрешено применять следующие модификаторы.

- Статический модификатор `static`.
- Модификаторы доступа: `public`, `internal`, `private`, `protected`.
- Модификатор наследования `new`.
- Модификатор небезопасного кода `unsafe`.
- Модификатор доступа только для чтения `readonly`.
- Модификатор многопоточности `volatile`.

### Модификатор `readonly`

Модификатор `readonly` предотвращает изменение поля после его создания. Присваивать значение полю, допускающему только чтение, можно только в его объявлении или внутри конструктора типа, где оно определено.

### Инициализация полей

Инициализация полей необязательна. Неинициализированное поле получает свое стандартное значение (`0`, `\0`, `null`, `false`). Инициализаторы полей выполняются перед конструкторами:

```
public int Age = 10;
```

### Объявление множества полей вместе

Для удобства множество полей одного типа можно объявлять в списке, разделяя их запятыми. Это подходящий способ обеспечить совместное использование всеми полями одних и тех же атрибутов и модификаторов полей. Например:

```
static readonly int legs = 8,
                    eyes = 2;
```

## Методы

Метод выполняет действие в виде последовательности операторов. Метод может получать *входные* данные из вызывающего кода посредством указания *параметров* и возвращать *выходные* данные обратно вызывающему коду за счет указания *возвращаемого типа*. Для метода может быть определен возвращаемый тип `void`, который говорит о том, что метод никакого значения не возвращает. Метод также может возвращать выходные данные вызывающему коду через параметры `ref` и `out`.

*Сигнатура* метода должна быть уникальной в рамках типа. Сигнатура метода включает в себя имя метода и типы параметров (но не содержит *имена* параметров и возвращаемый тип).

С методами разрешено применять следующие модификаторы.

- Статический модификатор `static`.
- Модификаторы доступа: `public`, `internal`, `private`, `protected`.
- Модификаторы наследования: `new`, `virtual`, `abstract`, `override`, `sealed`.
- Модификатор частичного метода `partial`.
- Модификаторы неуправляемого кода: `unsafe`, `extern`.
- Модификатор асинхронного кода `async`.

### Методы, сжатые до выражений (C# 6)

Метод, который состоит из единственного выражения, как показано ниже:

```
int Foo (int x) { return x * 2; }
```

можно записать более кратко как *метод, сжатый до выражения* (expression-bodied method). Фигурные скобки и ключевое слово `return` заменяются комбинацией =>:

```
int Foo (int x) => x * 2;
```

Функции, сжатые до выражений, могут также иметь возвращаемый тип `void`:

```
void Foo (int x) => Console.WriteLine (x);
```

### Перегрузка методов

Тип может перегружать методы (иметь несколько методов с одним и тем же именем) при условии, что их сигнатуры будут отличаться. Например, все перечисленные ниже методы могут сосуществовать внутри одного типа:

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (int x, float y) {...}
void Foo (float x, int y) {...}
```

Тем не менее, следующие пары методов не могут сосуществовать в рамках одного типа, поскольку возвращаемый тип и модификатор `params` не входят в состав сигнатуры метода:

```
void  Foo (int x) {...}
float Foo (int x) {...}            // Ошибка на этапе компиляции

void  Goo (int[] x) {...}
void  Goo (params int[] x) {...}   // Ошибка на этапе компиляции
```

### Передача по значению или передача по ссылке

Способ передачи параметра — по значению или по ссылке — также является частью сигнатуры. Например, `Foo(int)` может сосуществовать вместе с `Foo(ref int)` или `Foo(out int)`. Однако `Foo(ref int)` и `Foo(out int)` сосуществовать не могут:

```
void Foo (int x) {...}
void Foo (ref int x) {...}         // До этой точки все в порядке
void Foo (out int x) {...}         // Ошибка на этапе компиляции
```

### Локальные методы (C# 7)

В версии C# 7 можно определять метод внутри другого метода:

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  Console.WriteLine (Cube (4));
  Console.WriteLine (Cube (5));

  int Cube (int value) => value * value * value;
}
```

Локальный метод (`Cube()` в данном случае) будет видимым только для охватывающего метода (`WriteCubes()`). Это упрощает содержащий тип и немедленно подает сигнал любому просматривающему код, что `Cube` больше нигде не применяется. Еще одно преимущество локальных методов в том, что они могут обращаться к локальным

переменным и параметрам охватывающего метода. С такой особенностью связано несколько последствий, которые описаны в разделе "Захватывание внешних переменных" главы 4.

Локальные методы могут появляться внутри функций других видов, таких как средства доступа к свойствам, конструкторы и т.д. Локальные методы можно даже помещать внутрь других локальных методов и лямбда-выражений, которые используют блок операторов (глава 4). Локальные методы могут быть итераторными (глава 4) или асинхронными (глава 14).

Модификатор `static` для локальных методов недействителен. Они будут неявно статическими, если охватывающий метод является статическим.

## Конструкторы экземпляров

Конструкторы выполняют код инициализации класса или структуры. Конструктор определяется подобно методу за исключением того, что вместо имени метода и возвращаемого типа указывается имя типа, к которому относится этот конструктор:

```
public class Panda
{
  string name;                  // Определение поля
  public Panda (string n)       // Определение конструктора
  {
    name = n;                   // Код инициализации (установка поля)
  }
}
...
Panda p = new Panda ("Petey"); // Вызов конструктора
```

Конструкторы экземпляров допускают применение следующих модификаторов.

- Модификаторы доступа: `public`, `internal`, `private`, `protected`.
- Модификаторы неуправляемого кода: `unsafe`, `extern`.

В версии C# 7 конструкторы, содержащие единственный оператор, могут записываться как члены, сжатые до выражений:

```
public Panda (string n) => name = n;
```

### Перегрузка конструкторов

Класс или структура может перегружать конструкторы. Один перегруженный конструктор может вызывать другой, используя ключевое слово `this`:

```
using System;

public class Wine
{
  public decimal Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

Когда один конструктор вызывает другой, то первым выполняется *вызванный конструктор*.

Другому конструктору можно передавать *выражение*:

```
public Wine (decimal price, DateTime year) : this (price, year.Year) { }
```

В самом выражении не допускается применять ссылку this, скажем, для вызова метода экземпляра. (Причина в том, что на данной стадии объект еще не инициализирован конструктором, поэтому вызов любого метода вполне вероятно приведет к сбою.) Тем не менее, вызывать статические методы разрешено.

### Неявные конструкторы без параметров

Компилятор C# автоматически генерирует для класса открытый конструктор без параметров тогда и только тогда, когда в нем не было определено ни одного конструктора. Однако после определения хотя бы одного конструктора конструктор без параметров больше автоматически не генерируется.

### Порядок выполнения конструктора и инициализации полей

Как было показано ранее, поля могут инициализироваться стандартными значениями при их объявлении:

```
class Player
{
  int shields = 50;  // Инициализируется первым
  int health = 100;  // Инициализируется вторым
}
```

Инициализация полей происходит *перед* выполнением конструктора в порядке их объявления.

### Неоткрытые конструкторы

Конструкторы не обязательно должны быть открытыми. Распространенной причиной наличия неоткрытого конструктора является управление созданием экземпляров через вызов статического метода. Статический метод может использоваться для возвращения объекта из пула вместо создания нового объекта или для возвращения экземпляра специализированного подкласса, выбираемого на основе входных аргументов:

```
public class Class1
{
  Class1() {}  // Закрытый конструктор
  public static Class1 Create (...)
  {
    // Специальная логика для возвращения экземпляра Class1
    ...
  }
}
```

## Деконструкторы (C# 7)

В версии C# 7 появился шаблон *деконструирования*. Деконструктор (также называемый *методом деконструирования*) действует почти как противоположность конструктору: в то время когда конструктор обычно принимает набор значений (в качестве параметров) и присваивает их полям, деконструктор делает обратное, присваивая поля набору переменных.

Метод деконструирования должен называться Deconstruct и иметь один или большее число параметров out, как в следующем классе:

```
class Rectangle
{
  public readonly float Width, Height;

  public Rectangle (float width, float height)
  {
    Width = width;
    Height = height;
  }

  public void Deconstruct (out float width, out float height)
  {
    width = Width;
    height = Height;
  }
}
```

Для вызова деконструктора применяется следующий специальный синтаксис:

```
var rect = new Rectangle (3, 4);
(float width, float height) = rect;          // Деконструирование
Console.WriteLine (width + " " + height);    // 3 4
```

Деконструирующий вызов содержится во второй строке. Он создает две локальные переменные и затем обращается к методу `Deconstruct`. Вот чему эквивалентен этот деконструирующий вызов:

```
float width, height;
rect.Deconstruct (out width, out height);
```

Или:

```
rect.Deconstruct (out var width, out var height);
```

Деконструирующие вызовы допускают неявную типизацию, так что наш вызов можно было бы сократить следующим образом:

```
(var width, var height) = rect;
```

Либо просто:

```
var (width, height) = rect;
```

Если переменные, в которые производится деконструирование, уже определены, то типы вообще не указываются:

```
float width, height;
(width, height) = rect;
```

Такое действие называется *деконструирующим присваиванием.*

Перегружая метод `Deconstruct`, вызывающему коду можно предложить целый диапазон вариантов деконструирования.

Метод `Deconstruct` может быть расширяющим методом (см. раздел "Расширяющие методы" в главе 4). Это удобный прием, если вы хотите деконструировать типы, автором которых не являетесь.

## Инициализаторы объектов

Для упрощения инициализации объекта любые его доступные поля и свойства можно устанавливать с помощью *инициализатора объекта* прямо после создания. Например, рассмотрим приведенный далее класс:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny () {}
  public Bunny (string n) { Name = n; }
}
```

Используя инициализаторы объектов, создавать объекты `Bunny` можно следующим образом:

```
// Обратите внимание, что для конструкторов без параметров круглые
// скобки можно не указывать
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo")     { LikesCarrots=true, LikesHumans=false };
```

Код, конструирующий объекты `b1` и `b2`, в точности эквивалентен такому коду:

```
Bunny temp1 = new Bunny();  // temp1 - имя, сгенерированное компилятором
temp1.Name = "Bo";
temp1.LikesCarrots = true;
temp1.LikesHumans = false;
Bunny b1 = temp1;

Bunny temp2 = new Bunny ("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

Временные переменные гарантируют, что если в течение инициализации произойдет исключение, то вы в итоге не получите наполовину инициализированный объект.

### Сравнение инициализаторов объектов и необязательных параметров

Вместо применения инициализаторов объектов мы могли бы сделать так, чтобы конструктор класса `Bunny` принимал необязательные параметры:

```
public Bunny (string name,
              bool likesCarrots = false,
              bool likesHumans = false)
{
  Name = name;
  LikesCarrots = likesCarrots;
  LikesHumans = likesHumans;
}
```

Тогда появилась бы возможность конструировать экземпляр `Bunny` так, как показано ниже:

```
Bunny b1 = new Bunny (name: "Bo",
                      likesCarrots: true);
```

Преимущество данного подхода заключается в том, что при желании можно было бы сделать поля класса `Bunny` (или *свойства*, как вскоре будет объяснено) доступными только для чтения. Превращать поля или свойства в допускающие только чтение рекомендуется тогда, когда нет законного основания для их изменения в течение времени существования объекта.

Недостаток такого подхода состоит в том, что значение каждого необязательного параметра внедряется в *место вызова*. Другими словами, компилятор C# транслирует показанный выше вызов конструктора в следующий код:

```
Bunny b1 = new Bunny ("Bo", true, false);
```

Но если мы создаем экземпляр класса `Bunny` из другой сборки и позже модифицируем этот класс, добавив еще один необязательный параметр (скажем, `likesCats`), то могут возникнуть проблемы. Если не перекомпилировать ссылающуюся сборку, тогда она продолжит вызывать (уже несуществующий) конструктор с тремя параметрами, приводя к ошибке во время выполнения. (Более тонкая проблема связана с тем, что даже когда мы изменяем значение одного из необязательных параметров, вызывающий код в других сборках продолжит пользоваться старым необязательным значением до тех пор, пока эти сборки не будут перекомпилированы.)

Таким образом, если вы хотите поддерживать двоичную совместимость между версиями сборок, то должны уделять особое внимание необязательным параметрам в открытых функциях.

---

Инициализаторы объектов появились в версии C# 3.0.

## Ссылка `this`

Ссылка `this` указывает на сам экземпляр. В следующем примере метод `Marry` применяет ссылку `this` для установки поля `Mate` экземпляра `partner`:

```
public class Panda
{
  public Panda Mate;

  public void Marry (Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

Ссылка `this` также устраняет неоднозначность между локальной переменной или параметром и полем. Например:

```
public class Test
{
  string name;
  public Test (string name) { this.name = name; }
}
```

Использование ссылки `this` допускается только внутри нестатических членов класса или структуры.

## Свойства

Снаружи свойства выглядят похожими на поля, но внутренне они подобно методам содержат логику.

Например, взглянув на следующий код, невозможно сказать, чем является CurrentPrice – полем или свойством:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

Свойство объявляется похожим на поле образом, но с добавлением блока get/set. Ниже показано, как реализовать CurrentPrice в виде свойства:

```
public class Stock
{
  decimal currentPrice;            // Закрытое "поддерживающее" поле

  public decimal CurrentPrice      // Открытое свойство
  {
    get { return currentPrice; }
    set { currentPrice = value; }
  }
}
```

С помощью get и set обозначаются *средства доступа* к свойству. Средство доступа get запускается при чтении свойства. Оно должно возвращать значение, имеющее тип как у самого свойства. Средство доступа set выполняется во время присваивания свойству значения. Оно принимает неявный параметр по имени value с типом свойства, который обычно присваивается какому-то закрытому полю (в данном случае currentPrice).

Хотя доступ к свойствам осуществляется таким же способом, как и к полям, свойства отличаются тем, что предоставляют программисту полный контроль над получением и установкой их значений. Такой контроль позволяет программисту выбрать любое необходимое внутреннее представление, не открывая внутренние детали пользователю свойства. В приведенном примере метод set мог бы генерировать исключение, если значение value выходит за пределы допустимого диапазона.

В настоящей книге повсеместно применяются открытые поля, чтобы излишне не усложнять примеры и не отвлекать от их сути. В реальном приложении для содействия инкапсуляции предпочтение обычно отдается открытым свойствам, а не открытым полям.

Со свойствами разрешено применять следующие модификаторы.

- Статический модификатор static.
- Модификаторы доступа: public, internal, private, protected.
- Модификаторы наследования: new, virtual, abstract, override, sealed.
- Модификаторы неуправляемого кода: unsafe, extern.

### Свойства только для чтения и вычисляемые свойства

Свойство будет разрешать только чтение, если для него указано лишь средство доступа get, и только запись, если определено лишь средство доступа set. Свойства только для записи используются редко.

Свойство обычно имеет отдельное поддерживающее поле, предназначенное для хранения внутренних данных. Тем не менее, свойство может также возвращать значение, вычисленное на основе других данных. Например:

```
decimal currentPrice, sharesOwned;
public decimal Worth
{
  get { return currentPrice * sharesOwned; }
}
```

### Свойства, сжатые до выражений (C# 6, C# 7)

Начиная с версии C# 6, свойство только для чтения, такое как показанное в предыдущем примере, можно объявлять более кратко в виде *свойства, сжатого до выражения* (expression-bodied property). Все фигурные скобки и ключевые слова `get` и `return` заменяются комбинацией `=>`:

```
public decimal Worth => currentPrice * sharesOwned;
```

В версии C# 7 дополнительно допускается объявлять сжатыми до выражения также и средства доступа `set`:

```
public decimal Worth
{
  get => currentPrice * sharesOwned;
  set => sharesOwned = value / currentPrice;
}
```

### Автоматические свойства

Наиболее распространенная реализация свойства предусматривает наличие средств доступа `get` и/или `set`, которые просто читают и записывают в закрытое поле того же типа, что и свойство. Объявление *автоматического свойства* указывает компилятору на необходимость предоставления такой реализации. Первый пример в этом разделе можно усовершенствовать, объявив `CurrentPrice` как автоматическое свойство:

```
public class Stock
{
  ...
  public decimal CurrentPrice { get; set; }
}
```

Компилятор автоматически создает закрытое поддерживающее поле со специальным сгенерированным именем, ссылаться на которое невозможно. Средство доступа `set` может быть помечено как `private` или `protected`, если свойство должно быть доступно другим типам только для чтения. Автоматические свойства появились в версии C# 3.0.

### Инициализаторы свойств (C# 6)

Начиная с версии C# 6, к автоматическим свойствам можно добавлять инициализаторы свойств в точности как к полям:

```
public decimal CurrentPrice { get; set; } = 123;
```

В результате свойство `CurrentPrice` получает начальное значение `123`. Свойства с инициализаторами могут допускать только чтение:

```
public int Maximum { get; } = 999;
```

Как и поля, предназначенные только для чтения, автоматические свойства, допускающие только чтение, могут устанавливаться также в конструкторе типа. Это удобно при создании *неизменяемых* (только для чтения) типов.

### Доступность get и set

Средства доступа get и set могут иметь разные уровни доступа. В типичном сценарии применения есть свойство public с модификатором доступа internal или private, указанным для средства доступа set:

```
public class Foo
{
  private decimal x;
  public decimal X
  {
    get         { return x;  }
    private set { x = Math.Round (value, 2); }
  }
}
```

Обратите внимание, что само свойство объявлено с более либеральным уровнем доступа (public в данном случае), а к средству доступа, которое должно быть *менее* доступным, добавлен соответствующий модификатор.

### Реализация свойств в CLR

Средства доступа к свойствам C# внутренне компилируются в методы с именами get_*XXX* и set_*XXX*:

```
public decimal get_CurrentPrice {...}
public void set_CurrentPrice (decimal value) {...}
```

Простые невиртуальные средства доступа к свойствам *встраиваются* компилятором JIT, устраняя любую разницу в производительности между доступом к свойству и доступом к полю. Встраивание – это разновидность оптимизации, при которой вызов метода заменяется телом этого метода.

Для свойств WinRT компилятор предполагает применение соглашения об именовании put_*XXX*, а не set_*XXX*.

## Индексаторы

Индексаторы предоставляют естественный синтаксис для доступа к элементам в классе или структуре, которая инкапсулирует список либо словарь значений. Индексаторы похожи на свойства, но предусматривают доступ через аргумент индекса, а не через имя свойства. Класс string имеет индексатор, который позволяет получать доступ к каждому его значению char посредством индекса int:

```
string s = "hello";
Console.WriteLine (s[0]);  // 'h'
Console.WriteLine (s[3]);  // 'l'
```

Синтаксис использования индексаторов похож на синтаксис работы с массивами за исключением того, что аргумент (аргументы) индекса может быть любого типа (типов).

Индексаторы имеют те же модификаторы, что и свойства (см. раздел "Свойства" ранее в главе), и могут вызываться null-условным образом за счет помещения вопросительного знака перед открывающей квадратной скобкой (см. раздел "Операции для работы со значениями null" в главе 2):

```
string s = null;
Console.WriteLine (s?[0]);  // Ничего не выводится; ошибка не возникает
```

### Реализация индексатора

Для реализации индексатора понадобится определить свойство по имени `this`, указав аргументы в квадратных скобках. Например:

```
class Sentence
{
  string[] words = "The quick brown fox".Split();
  public string this [int wordNum]  // индексатор
  {
    get { return words [wordNum]; }
    set { words [wordNum] = value; }
  }
}
```

Ниже показано, как можно было бы применять такой индексатор:

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);     // fox
s[3] = "kangaroo";
Console.WriteLine (s[3]);     // kangaroo
```

Для типа можно объявлять несколько индексаторов, каждый с параметрами разных типов. Индексатор также может принимать более одного параметра:

```
public string this [int arg1, string arg2]
{
  get { ... }  set { ... }
}
```

Если опустить средство доступа `set`, то индексатор станет предназначенным только для чтения, а начиная с версии C# 6, его определение можно сократить с помощью синтаксиса, сжатого до выражения:

```
public string this [int wordNum] => words [wordNum];
```

### Реализация индексаторов в CLR

Индексаторы внутренне компилируются в методы с именами `get_Item` и `set_Item`, как показано ниже:

```
public string get_Item (int wordNum) {...}
public void set_Item (int wordNum, string value) {...}
```

## Константы

*Константа* – это статическое поле, значение которого никогда не может изменяться. Константа оценивается статически на этапе компиляции и компилятор литеральным образом подставляет ее значение всегда, когда она встречается (довольно похоже на макрос в C++). Константа может относиться к любому из встроенных числовых типов, `bool`, `char`, `string` или `enum`.

Константа объявляется с использованием ключевого слова `const` и должна быть инициализирована каким-нибудь значением. Например:

```
public class Test
{
  public const string Message = "Hello World";
}
```

Константа гораздо более ограничена, чем поле `static readonly` — как в типах, которые можно применять, так и в семантике инициализации поля. Константа также отличается от поля `static readonly` тем, что ее оценка производится на этапе компиляции. Например:

```
public static double Circumference (double radius)
{
  return 2 * System.Math.PI * radius;
}
```

компилируется в:

```
public static double Circumference (double radius)
{
  return 6.2831853071795862 * radius;
}
```

Для `PI` имеет смысл быть константой, т.к. значение никогда не меняется. Напротив, поле `static readonly` может иметь отличающееся значение в каждом приложении.

Поле `static readonly` также полезно, когда другим сборкам открывается доступ к значению, которое в более поздней версии сборки может измениться. Например, пусть сборка `X` открывает доступ к константе, как показано ниже:

```
public const decimal ProgramVersion = 2.3;
```

Если сборка `Y` ссылается на сборку `X` и пользуется константой `ProgramVersion`, тогда при компиляции значение `2.3` будет встроено в сборку `Y`. В результате, когда сборка `X` позже перекомпилируется с константой `ProgramVersion`, установленной в `2.4`, в сборке `Y` по-прежнему будет применяться старое значение `2.3` *до тех пор, пока сборка Y не будет перекомпилирована.* Поле `static readonly` позволяет избежать такой проблемы.

На описанную ситуацию можно взглянуть и по-другому: любое значение, которое способно измениться в будущем, не является константой по определению, а потому не должно быть представлено в виде константы.

Константы можно также объявлять локально внутри метода. Например:

```
static void Main()
{
  const double twoPI  = 2 * System.Math.PI;
  ...
}
```

Нелокальные константы допускают использование следующих модификаторов.

- Модификаторы доступа: `public`, `internal`, `private`, `protected`.
- Модификатор наследования `new`.

## Статические конструкторы

Статический конструктор выполняется однократно для *типа*, а не однократно для *экземпляра*. В типе может быть определен только один статический конструктор, он должен не принимать параметры и иметь то же имя, что и тип:

```
class Test
{
  static Test() { Console.WriteLine ("Type Initialized"); }
}
```

Исполняющая среда автоматически вызывает статический конструктор прямо перед тем, как тип начинает применяться. Вызов инициируется двумя действиями:

- создание экземпляра типа;
- доступ к статическому члену типа.

Для статических конструкторов разрешены только модификаторы `unsafe` и `extern`.

Если статический конструктор генерирует необработанное исключение (глава 4), то тип, к которому он относится, становится *непригодным* в жизненном цикле приложения.

### Статические конструкторы и порядок инициализации полей

Инициализаторы статических полей запускаются непосредственно *перед* вызовом статического конструктора. Если тип не имеет статического конструктора, тогда инициализаторы полей будут выполняться до того, как тип начнет использоваться — или *в любой момент раньше* по прихоти исполняющей среды.

Инициализаторы статических полей выполняются в порядке объявления полей. Сказанное демонстрируется в следующем примере: поле `X` инициализируется значением `0`, а поле `Y` — значением `3`:

```
class Foo
{
  public static int X = Y;  // 0
  public static int Y = 3;  // 3
}
```

Если поменять местами эти два инициализатора полей, то оба поля будут инициализированы значением `3`. В показанном ниже примере на экран выводится `0`, а затем `3`, т.к. инициализатор поля, который создает экземпляр `Foo`, выполняется до того, как поле `X` инициализируется значением `3`:

```
class Program
{
  static void Main() { Console.WriteLine (Foo.X); }   // 3
}

class Foo
{
  public static Foo Instance = new Foo();
  public static int X = 3;

  Foo() { Console.WriteLine (X); }                    // 0
}
```

Если поменять местами строки кода, выделенные полужирным, тогда на экран будет выводиться `3` и затем снова `3`.

## Статические классы

Класс может быть помечен как `static`, указывая на то, что он должен состоять исключительно из статических членов и не допускать создание подклассов на своей основе. Хорошими примерами статических классов могут служить `System.Console` и `System.Math`.

## Финализаторы

Финализаторы представляют собой методы, предназначенные только для классов, которые выполняются до того, как сборщик мусора освободит память, занятую объектом с отсутствующими ссылками на него. Синтаксически финализатор записывается как имя класса, предваренное символом ~:

```
class Class1
{
  ~Class1()
  {
    ...
  }
}
```

В действительности это синтаксис C# для переопределения метода `Finalize` класса `Object`, и компилятор развертывает его в следующее объявление метода:

```
protected override void Finalize()
{
  ...
  base.Finalize();
}
```

Сборка мусора и финализаторы подробно обсуждаются в главе 12.

Финализаторы допускают применение следующего модификатора.

- Модификатор неуправляемого кода `unsafe`.

В версии C# 7 финализаторы, состоящие из единственного оператора, могут быть записаны с помощью синтаксиса сжатия до выражения:

```
~Class1() => Console.WriteLine ("Finalizing");
```

## Частичные типы и методы

Частичные типы позволяют расщеплять определение типа, обычно разнося его по нескольким файлам. Распространенный сценарий предполагает автоматическую генерацию частичного класса из какого-то другого источника (например, шаблона Visual Studio) и последующее его дополнение вручную написанными методами. Например:

```
// PaymentFormGen.cs - сгенерирован автоматически
partial class PaymentForm { ... }

// PaymentForm.cs - написан вручную
partial class PaymentForm { ... }
```

Каждый участник должен иметь объявление `partial`; показанный ниже код является недопустимым:

```
partial class PaymentForm {}
class PaymentForm {}
```

Участники не могут содержать конфликтующие члены. Скажем, конструктор с теми же самыми параметрами повторять нельзя. Частичные типы распознаются полностью компилятором, а это значит, что каждый участник должен быть доступным на этапе компиляции и располагаться в той же сборке.

Базовый класс может быть указан для единственного участника или для множества участников (при условии, что для каждого из них базовый класс будет тем же самым). Кроме того, для каждого участника можно независимо указывать интерфейсы,

подлежащие реализации. Базовые классы и интерфейсы рассматриваются в разделах "Наследование" и "Интерфейсы" далее в главе.

Компилятор не гарантирует какого-то определенного порядка инициализации полей в рамках объявлений частичных типов.

### Частичные методы

Частичный тип может содержать *частичные методы*. Они позволяют автоматически сгенерированному частичному типу предоставлять настраиваемые точки привязки для ручного написания кода. Например:

```
partial class PaymentForm     // В автоматически сгенерированном файле
{
  ...
  partial void ValidatePayment (decimal amount);
}
partial class PaymentForm     // В написанном вручную файле
{
  ...
  partial void ValidatePayment (decimal amount)
  {
    if (amount > 100)
      ...
  }
}
```

Частичный метод состоит из двух частей: *определение* и *реализация*. Определение обычно записывается генератором кода, а реализация – вручную. Если реализация не предоставлена, тогда определение частичного метода при компиляции удаляется (вместе с кодом, в котором он вызывается). Это дает автоматически сгенерированному коду большую свободу в предоставлении точек привязки, не заставляя беспокоиться по поводу эффекта разбухания кода. Частичные методы должны быть `void`, и они неявно являются `private`.

Частичные методы появились в версии C# 3.0.

## Операция `nameof` (C# 6)

Операция `nameof` возвращает имя любого символа (типа, члена, переменной и т.д.) в виде строки:

```
int count = 123;
string name = nameof (count);  // name получает значение "count"
```

Преимущество использования данной операции по сравнению с простым указанием строки связано со статической проверкой типов. Инструменты, подобные Visual Studio, способны воспринимать символические ссылки, поэтому переименование любого символа приводит к переименованию также всех ссылок на него.

Для указания имени члена типа, такого как поле или свойство, необходимо включать тип члена. Это работает со статическими членами и членами экземпляра:

```
string name = nameof (StringBuilder.Length);
```

Результатом будет `"Length"`. Чтобы возвратить `"StringBuilder.Length"`, понадобится следующее выражение:

```
nameof(StringBuilder)+"."+nameof(StringBuilder.Length);
```

# Наследование

Класс может быть *унаследован* от другого класса с целью расширения или настройки исходного класса. Наследование от класса позволяет повторно использовать функциональность данного класса вместо ее построения с нуля. Класс может наследоваться только от одного класса, но сам может быть унаследован множеством классов, формируя иерархию классов. В этом примере мы начнем с определения класса по имени `Asset`:

```
public class Asset
{
  public string Name;
}
```

Далее мы определим классы `Stock` и `House`, которые будут унаследованы от `Asset`. Классы `Stock` и `House` получат все, что имеет `Asset`, плюс любые дополнительные члены, которые в них будут определены:

```
public class Stock : Asset  // унаследован от Asset
{
  public long SharesOwned;
}

public class House : Asset  // унаследован от Asset
{
  public decimal Mortgage;
}
```

Вот как можно работать с данными классами:

```
Stock msft = new Stock { Name="MSFT",
                         SharesOwned=1000 };

Console.WriteLine (msft.Name);              // MSFT
Console.WriteLine (msft.SharesOwned);       // 1000

House mansion = new House { Name="Mansion",
                            Mortgage=250000 };

Console.WriteLine (mansion.Name);           // Mansion
Console.WriteLine (mansion.Mortgage);       // 250000
```

*Производные классы* `Stock` и `House` наследуют свойство `Name` от *базового класса* `Asset`.

Производный класс также называется *подклассом*.

Базовый класс также называется *суперклассом*.

## Полиморфизм

Ссылки являются полиморфными. Это значит, что переменная типа `x` может ссылаться на объект, относящийся к подклассу `x`. Например, рассмотрим следующий метод:

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

Метод Display способен отображать значение свойства Name объектов Stock и House, т.к. они оба являются Asset:

```
Stock msft    = new Stock ... ;
House mansion = new House ... ;

Display (msft);
Display (mansion);
```

В основе работы полиморфизма лежит тот факт, что подклассы (Stock и House) обладают всеми характеристиками своего базового класса (Asset). Однако обратное утверждение не верно. Если метод Display переписать так, чтобы он принимал House, то передавать ему Asset будет невозможно:

```
static void Main() { Display (new Asset()); }  // Ошибка на этапе компиляции
public static void Display (House house)       // Asset приниматься не будет
{
  System.Console.WriteLine (house.Mortgage);
}
```

## Приведение и ссылочные преобразования

Ссылка на объект может быть:

- неявно *приведена вверх* к ссылке на базовый класс;
- явно *приведена вниз* к ссылке на подкласс.

Приведение вверх и вниз между совместимыми ссылочными типами выполняет *ссылочные преобразования*: создается новая ссылка, которая указывает на *тот же самый* объект. Приведение вверх всегда успешно; приведение вниз успешно только в случае, когда объект подходящим образом типизирован.

### Приведение вверх

Операция приведения вверх создает ссылку на базовый класс из ссылки на подкласс. Например:

```
Stock msft = new Stock();
Asset a = msft;  // Приведение вверх
```

После приведения вверх переменная a по-прежнему ссылается на тот же самый объект Stock, что и переменная msft. Сам объект, на который имеются ссылки, не изменяется и не преобразуется:

```
Console.WriteLine (a == msft);       // True
```

Хотя переменные a и msft ссылаются на один и тот же объект, a обеспечивает более ограниченное представление этого объекта:

```
Console.WriteLine (a.Name);          // Нормально
Console.WriteLine (a.SharesOwned);   // Ошибка: член SharesOwned не определен
```

Последняя строка кода вызывает ошибку на этапе компиляции, поскольку переменная a имеет тип Asset несмотря на то, что она ссылается на объект типа Stock. Чтобы получить доступ к полю SharesOwned, экземпляр Asset потребуется *привести вниз* к Stock.

### Приведение вниз

Операция приведения вниз создает ссылку на подкласс из ссылки на базовый класс. Например:

```
Stock msft = new Stock();
Asset a = msft;                          // Приведение вверх
Stock s = (Stock)a;                      // Приведение вниз
Console.WriteLine (s.SharesOwned);       // Ошибка не возникает
Console.WriteLine (s == a);              // True
Console.WriteLine (s == msft);           // True
```

Как и в случае приведения вверх, затрагиваются только ссылки, но не лежащий в основе объект. Приведение вниз требует явного указания, потому что потенциально оно может не достичь успеха во время выполнения:

```
House h = new House();
Asset a = h;          // Приведение вверх всегда успешно
Stock s = (Stock)a;  // Ошибка приведения вниз: a не является Stock
```

Когда приведение вниз терпит неудачу, генерируется исключение `InvalidCastException`. Это пример *проверки типов во время выполнения*, которая более подробно рассматривается в разделе "Статическая проверка типов и проверка типов во время выполнения" далее в главе.

### Операция as

Операция `as` выполняет приведение вниз, которое в случае неудачи вычисляется как `null` (вместо генерации исключения):

```
Asset a = new Asset();
Stock s = a as Stock;  // s равно null; исключение не генерируется
```

Операция удобна, когда нужно организовать последующую проверку результата на предмет `null`:

```
if (s != null) Console.WriteLine (s.SharesOwned);
```

В отсутствие проверки подобного рода приведение удобно тем, что в случае его неудачи генерируется более полезное исключение. Мы можем проиллюстрировать сказанное, сравнив следующие две строки кода:

```
int shares = ((Stock)a).SharesOwned;    // Подход #1
int shares = (a as Stock).SharesOwned;  // Подход #2
```

Если `a` не является `Stock`, тогда первая строка кода сгенерирует исключение `InvalidCastException`, которое обеспечит точное описание того, что пошло не так. Вторая строка кода сгенерирует исключение `NullReferenceException`, которое не даст однозначного ответа на вопрос, была ли переменная `a` не `Stock` или же просто `a` была равна `null`.

На описанную ситуацию можно взглянуть и по-другому: посредством операции приведения вы сообщаете компилятору о том, что *уверены* в типе заданного значения; если это не так, тогда в коде присутствует ошибка, поэтому нужно сгенерировать исключение. С другой стороны, в случае операции `as` вы не уверены в типе значения и хотите организовать ветвление в соответствие с результатом во время выполнения.

Операция `as` не может выполнять *специальные преобразования* (см. раздел "Перегрузка операций" в главе 4), равно как и числовые преобразования.

```
long x = 3 as long;  // Ошибка на этапе компиляции
```

Операция as и операции приведения также будут выполнять приведения вверх, хотя это не особенно полезно, поскольку всю необходимую работу сделает неявное преобразование.

### Операция is

Операция is проверяет, будет ли преобразование ссылки успешным; другими словами, является ли объект производным от указанного класса (или реализует ли какой-то интерфейс). Она часто применяется при проверке перед приведением вниз:

```
if (a is Stock)
  Console.WriteLine (((Stock)a).SharesOwned);
```

Операция is также в результате дает true, если может успешно выполниться *распаковывающее преобразование* (см. раздел "Тип object" далее в главе). Тем не менее, она не принимает во внимание специальные или числовые преобразования.

### Операция is и шаблонные переменные (C# 7)

В версии C# 7 во время использования операции is можно вводить переменную:

```
if (a is Stock s)
  Console.WriteLine (s.SharesOwned);
```

Такой код эквивалентен следующему коду:

```
Stock s;
if (a is Stock)
{
  s = (Stock) a;
  Console.WriteLine (s.SharesOwned);
}
```

Введенная переменная доступна для "немедленного" потребления, поэтому показанный ниже код законен:

```
if (a is Stock s && s.SharesOwned > 100000)
  Console.WriteLine ("Wealthy");
```

И она остается в области видимости за пределами выражения is, давая возможность записывать так:

```
if (a is Stock s && s.SharesOwned > 100000)
  Console.WriteLine ("Wealthy");
else
  s = new Stock();                    // s в области видимости
Console.WriteLine (s.SharesOwned);    // s все еще в области видимости
```

## Виртуальные функции-члены

Функция, помеченная как *виртуальная* (virtual), может быть *переопределена* в подклассах, где требуется предоставление ее специализированной реализации. Объявлять виртуальными можно методы, индексаторы и события:

```
public class Asset
{
  public string Name;
  public virtual decimal Liability => 0;  // Свойство, сжатое до выражения
}
```

(Конструкция `Liability => 0` является сокращенной записью для `{ get { return 0; } }`. За дополнительной информацией по такому синтаксису обращайтесь в раздел "Свойства, сжатые до выражений (C# 6, C# 7)" ранее в главе.) Виртуальный метод переопределяется в подклассе с применением модификатора `override`:

```
public class Stock : Asset
{
  public long SharesOwned;
}
public class House : Asset
{
  public decimal Mortgage;
  public override decimal Liability => Mortgage;
}
```

По умолчанию свойство `Liability` класса `Asset` возвращает `0`. Класс `Stock` не нуждается в специализации этого поведения. Однако класс `House` специализирует свойство `Liability`, чтобы возвращать значение `Mortgage`:

```
House mansion = new House { Name="McMansion", Mortgage=250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability);      // 250000
Console.WriteLine (a.Liability);            // 250000
```

Сигнатуры, возвращаемые типы и доступность виртуального и переопределенного методов должны быть идентичными. Внутри переопределенного метода можно вызывать его реализацию из базового класса с помощью ключевого слова `base` (см. раздел "Ключевое слово base" далее в главе).

Вызов виртуальных методов внутри конструктора потенциально опасен, т.к. авторы подклассов при переопределении метода вряд ли знают о том, что работают с частично инициализированным объектом. Другими словами, переопределяемый метод может в итоге обращаться к методам или свойствам, зависящим от полей, которые пока еще не инициализированы конструктором.

## Абстрактные классы и абстрактные члены

Класс, объявленный как *абстрактный* (`abstract`), не разрешает создавать свои экземпляры. Взамен можно создавать только экземпляры его конкретных *подклассов*.

В абстрактных классах есть возможность определять *абстрактные члены*. Абстрактные члены похожи на виртуальные члены за исключением того, что они не предоставляют стандартную реализацию. Реализация должна обеспечиваться подклассом, если только подкласс тоже не объявлен как `abstract`:

```
public abstract class Asset
{
  // Обратите внимание на пустую реализацию
  public abstract decimal NetValue { get; }
}
public class Stock : Asset
{
  public long SharesOwned;
  public decimal CurrentPrice;
  // Переопределить подобно виртуальному методу
  public override decimal NetValue => CurrentPrice * SharesOwned;
}
```

## Сокрытие унаследованных членов

В базовом классе и подклассе могут быть определены идентичные члены. Например:

```
public class A      { public int Counter = 1; }
public class B : A  { public int Counter = 2; }
```

Говорят, что поле `Counter` в классе `B` *скрывает* поле `Counter` в классе `A`. Обычно это происходит случайно, когда член добавляется в базовый тип *после* того, как идентичный член был добавлен к подтипу. В таком случае компилятор генерирует предупреждение и затем разрешает неоднозначность следующим образом:

- ссылки на `A` (на этапе компиляции) привязываются к `A.Counter`;
- ссылки на `B` (на этапе компиляции) привязываются к `B.Counter`.

Иногда необходимо преднамеренно сокрыть какой-то член; тогда к члену в подклассе можно применить ключевое слово `new`. Модификатор `new` *не делает ничего сверх того, что просто подавляет выдачу компилятором соответствующего предупреждения*:

```
public class A     { public     int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

Модификатор `new` сообщает компилятору — и другим программистам — о том, что дублирование члена произошло не случайно.

Ключевое слово `new` в языке C# перегружено и в разных контекстах имеет независимый смысл. В частности, *операция* `new` отличается от *модификатора членов* `new`.

### Сравнение `new` и `override`

Рассмотрим следующую иерархию классов:

```
public class BaseClass
{
  public virtual void Foo()  { Console.WriteLine ("BaseClass.Foo"); }
}
public class Overrider : BaseClass
{
  public override void Foo() { Console.WriteLine ("Overrider.Foo"); }
}
public class Hider : BaseClass
{
  public new void Foo()      { Console.WriteLine ("Hider.Foo"); }
}
```

Ниже приведен код, демонстрирующий отличия в поведении классов `Overrider` и `Hider`:

```
Overrider over = new Overrider();
BaseClass b1 = over;
over.Foo();          // Overrider.Foo
b1.Foo();            // Overrider.Foo

Hider h = new Hider();
BaseClass b2 = h;
h.Foo();             // Hider.Foo
b2.Foo();            // BaseClass.Foo
```

## Запечатывание функций и классов

С помощью ключевого слова `sealed` переопределенная функция может *запечатывать* свою реализацию, предотвращая ее переопределение другими подклассами. В ранее показанном примере виртуальной функции-члена можно было бы запечатать реализацию `Liability` в классе `House`, чтобы запретить переопределение `Liability` в классе, производном от `House`:

```
public sealed override decimal Liability { get { return Mortgage; } }
```

Можно также запечатать весь класс, неявно запечатав все его виртуальные функции, путем применения модификатора `sealed` к самому классу. Запечатывание класса встречается чаще, чем запечатывание функции-члена.

Хотя можно запечатывать, предотвращая переопределение, нет возможности запечатывать с целью предотвращения *сокрытия*.

## Ключевое слово `base`

Ключевое слово `base` похоже на ключевое слово `this`. Оно служит двум важным целям:

- доступ к функции-члену базового класса при ее переопределении в подклассе;
- вызов конструктора базового класса (см. следующий раздел).

В приведенном ниже примере в классе `House` ключевое слово `base` используется для доступа к реализации `Liability` из `Asset`:

```
public class House : Asset
{
  ...
  public override decimal Liability => base.Liability + Mortgage;
}
```

С помощью ключевого слова `base` мы получаем доступ к свойству `Liability` класса `Asset` *невиртуальным* образом. Это значит, что мы всегда обращаемся к версии `Asset` данного свойства независимо от действительного типа экземпляра во время выполнения.

Тот же самый подход работает в ситуации, когда `Liability` *скрывается*, а не *переопределяется*. (Получить доступ к скрытым членам можно также путем приведения к базовому классу перед обращением к члену.)

## Конструкторы и наследование

В подклассе должны быть объявлены собственные конструкторы. Конструкторы базового класса *доступны* в производном классе, но они никогда автоматически не *наследуются*. Например, если классы `Baseclass` и `Subclass` определены следующим образом:

```
public class Baseclass
{
  public int X;
  public Baseclass () { }
  public Baseclass (int x) { this.X = x; }
}
public class Subclass : Baseclass { }
```

то приведенный ниже код будет недопустимым:

```
Subclass s = new Subclass (123);
```

Следовательно, в классе Subclass должны быть "повторно определены" любые конструкторы, к которым необходимо открыть доступ. Тем не менее, с применением ключевого слова base можно вызывать любой конструктор базового класса:

```
public class Subclass : Baseclass
{
  public Subclass (int x) : base (x) { }
}
```

Ключевое слово base работает подобно ключевому слову this, но только вызывает конструктор базового класса.

Конструкторы базового класса всегда выполняются первыми, что гарантирует выполнение *базовой* инициализации перед *специализированной* инициализацией.

### Неявный вызов конструктора без параметров базового класса

Если в конструкторе подкласса опустить ключевое слово base, то будет неявно вызываться конструктор *без параметров* базового класса:

```
public class BaseClass
{
  public int X;
  public BaseClass() { X = 1; }
}
public class Subclass : BaseClass
{
  public Subclass()  { Console.WriteLine (X); }  // 1
}
```

Если базовый класс не имеет доступного конструктора без параметров, тогда в конструкторах подклассов придется использовать ключевое слово base.

### Конструктор и порядок инициализации полей

Когда объект создан, инициализация происходит в указанном ниже порядке.

1. От подкласса к базовому классу:
   а) инициализируются поля;
   б) оцениваются аргументы для вызова конструкторов базового класса.
2. От базового класса к подклассу:
   а) выполняются тела конструкторов.

Порядок демонстрируется в следующем коде:

```
public class B
{
  int x = 1;          // Выполняется третьим
  public B (int x)
  {
    ...               // Выполняется четвертым
  }
}
public class D : B
{
  int y = 1;          // Выполняется первым
```

```
  public D (int x)
    : base (x + 1)     // Выполняется вторым
  {
    ...                // Выполняется пятым
  }
}
```

## Перегрузка и распознавание

Наследование оказывает интересное влияние на перегрузку методов. Предположим, что есть следующие две перегруженные версии:

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

При вызове перегруженной версии приоритет получает наиболее специфичный тип:

```
House h = new House (...);
Foo(h);  // Вызывается Foo(House)
```

Конкретная перегруженная версия, подлежащая вызову, определяется статически (на этапе компиляции), а не во время выполнения. В показанном ниже коде вызывается `Foo(Asset)` несмотря на то, что типом времени выполнения переменной `a` является `House`:

```
Asset a = new House (...);
Foo(a);  // Вызывается Foo(Asset)
```

Если привести `Asset` к `dynamic` (глава 4), тогда решение о том, какая перегруженная версия должна вызываться, откладывается до стадии выполнения, и выбор будет основан на действительном типе объекта:

```
Asset a = new House (...);
Foo ((dynamic)a);  // Вызывается Foo(House)
```

# Тип object

Тип `object` (`System.Object`) представляет собой первоначальный базовый класс для всех типов. Любой тип может быть неявно приведен вверх к `object`.

Чтобы проиллюстрировать, насколько это полезно, рассмотрим универсальный *стек*. Стек является структурой данных, работа которой основана на принципе LIFO ("Last-In First-Out" – "последним пришел – первым обслужен"). Стек поддерживает две операции: *заталкивание* объекта в стек и *выталкивание* объекта из стека. Ниже показана простая реализация, которая способна хранить до 10 объектов:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj)  { data[position++] = obj;  }
  public object Pop()            { return data[--position]; }
}
```

Поскольку `Stack` работает с типом `object`, методы `Push()` и `Pop()` класса `Stack` можно применять с экземплярами *любого типа*:

```
Stack stack = new Stack();
stack.Push ("sausage");
string s = (string) stack.Pop();   // Приведение вниз должно быть явным
Console.WriteLine (s);             // Выводит sausage
```

`object` относится к ссылочным типам в силу того, что представляет собой класс. Несмотря на данное обстоятельство, типы значений, такие как `int`, также можно приводить к `object`, а `object` приводить к ним, а потому они могут быть помещены в стек. Такая особенность C# называется *унификацией типов* и демонстрируется ниже:

```
stack.Push (3);
int three = (int) stack.Pop();
```

Когда запрашивается приведение между типом значения и типом `object`, среда CLR должна выполнить специальную работу по преодолению семантических отличий между типами значений и ссылочными типами. Процессы называются *упаковкой* (boxing) и *распаковкой* (unboxing).

В разделе "Обобщения" далее в главе будет показано, как усовершенствовать класс `Stack`, чтобы улучшить поддержку стеков однотипных элементов.

## Упаковка и распаковка

Упаковка представляет собой действие по приведению экземпляра типа значения к экземпляру ссылочного типа. Ссылочным типом может быть либо класс `object`, либо интерфейс (см. раздел "Интерфейсы" далее в главе)[1]. В следующем примере мы упаковываем `int` в объект:

```
int x = 9;
object obj = x;          // Упаковать int
```

Распаковка является обратной операцией, которая предусматривает приведение объекта к исходному типу значения:

```
int y = (int)obj;        // Распаковать int
```

Распаковка требует явного приведения. Исполняющая среда проверяет, соответствует ли указанный тип значения действительному объектному типу, и генерирует исключение `InvalidCastException`, если это не так. Например, показанный далее код приведет к генерации исключения, поскольку `long` не точно соответствует `int`:

```
object obj = 9;          // Для значения 9 выводится тип int
long x = (long) obj;     // Генерируется исключение InvalidCastException
```

Однако следующий код выполняется успешно:

```
object obj = 9;
long x = (int) obj;
```

Этот код также не вызывает ошибки:

```
object obj = 3.5;             // Для значения 3.5 выводится тип double
int x = (int) (double) obj;   // x теперь равно 3
```

В последнем примере `(double)` осуществляет *распаковку*, после чего `(int)` выполняет *числовое преобразование*.

[1] Ссылочным типом может также быть `System.ValueType` или `System.Enum` (глава 6).

Упаковывающие преобразования критически важны в обеспечении унифицированной системы типов. Тем не менее, эта система не идеальна: в разделе "Обобщения" далее в главе будет показано, что вариантность массивов и обобщений поддерживает только *ссылочные преобразования*, но не *упаковывающие преобразования*:

```
object[] a1 = new string[3];   // Допустимо
object[] a2 = new int[3];      // Ошибка
```

### Семантика копирования при упаковке и распаковке

Упаковка *копирует* экземпляр типа значения в новый объект, а распаковка *копирует* содержимое данного объекта обратно в экземпляр типа значения. В приведенном ниже примере изменение значения `i` не вызывает изменения ранее упакованной копии:

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine (boxed);  // 3
```

## Статическая проверка типов и проверка типов во время выполнения

Программы на языке C# подвергаются проверке типов как статически (на этапе компиляции), так и во время выполнения (средой CLR).

Статическая проверка типов позволяет компилятору контролировать корректность программы, не выполняя ее. Показанный ниже код не скомпилируется, т.к. компилятор принудительно применяет статическую проверку типов:

```
int x = "5";
```

Проверка типов во время выполнения осуществляется средой CLR, когда происходит приведение вниз через ссылочное преобразование или распаковку:

```
object y = "5";
int z = (int) y;  // Ошибка времени выполнения, неудача приведения вниз
```

Проверка типов во время выполнения возможна потому, что каждый объект в куче внутренне хранит небольшой маркер типа. Данный маркер может быть извлечен посредством вызова метода `GetType` класса `object`.

## Метод `GetType` и операция `typeof`

Все типы в C# во время выполнения представлены с помощью экземпляра `System.Type`. Получить объект `System.Type` можно двумя основными путями:

- вызвать метод `GetType` на экземпляре;
- воспользоваться операцией `typeof` на имени типа.

Результат `GetType` оценивается во время выполнения, а `typeof` — статически на этапе компиляции (когда вовлечены параметры обобщенных типов, они распознаются компилятором JIT).

В классе `System.Type` предусмотрены свойства для имени типа, сборки, базового типа и т.д. Например:

```
using System;

public class Point { public int X, Y; }
```

```
class Test
{
  static void Main()
  {
    Point p = new Point();
    Console.WriteLine (p.GetType().Name);                  // Point
    Console.WriteLine (typeof (Point).Name);               // Point
    Console.WriteLine (p.GetType() == typeof(Point));      // True
    Console.WriteLine (p.X.GetType().Name);                // Int32
    Console.WriteLine (p.Y.GetType().FullName);            // System.Int32
  }
}
```

В `System.Type` также имеются методы, которые действуют в качестве шлюза для модели рефлексии времени выполнения, которая описана в главе 19.

## Метод `ToString`

Метод `ToString` возвращает стандартное текстовое представление экземпляра типа. Данный метод переопределяется всеми встроенными типами. Ниже приведен пример применения метода `ToString` типа `int`:

```
int x = 1;
string s = x.ToString();  // s равно "1"
```

Переопределять метод `ToString` в специальных типах можно следующим образом:

```
public class Panda
{
  public string Name;
  public override string ToString() => Name;
}
...
Panda p = new Panda { Name = "Petey" };
Console.WriteLine (p);  // Petey
```

Если метод `ToString` не переопределять, то он будет возвращать имя типа.

В случае вызова *переопределенного* члена класса `object`, такого как `ToString`, непосредственно на типе значения упаковка не производится. Упаковка происходит позже, только если осуществляется приведение:

```
int x = 1;
string s1 = x.ToString();    // Вызывается на неупакованном значении
object box = x;
string s2 = box.ToString();  // Вызывается на упакованном значении
```

## Список членов `object`

Ниже приведен список всех членов `object`:

```
public class Object
{
  public Object();

  public extern Type GetType();

  public virtual bool Equals (object obj);
  public static bool Equals  (object objA, object objB);
```

```
  public static bool ReferenceEquals (object objA, object objB);
  public virtual int GetHashCode();
  public virtual string ToString();
  protected virtual void Finalize();
  protected extern object MemberwiseClone();
}
```

Методы `Equals`, `ReferenceEquals` и `GetHashCode` будут описаны в разделе "Сравнение эквивалентности" главы 6.

# Структуры

*Структура* похожа на класс, но обладает указанными ниже ключевыми отличиями.

- Структура является типом значения, тогда как класс – ссылочным типом.
- Структура не поддерживает наследование (за исключением того, что она неявно порождена от `object`, или точнее – от `System.ValueType`).

Структура может иметь все те же члены, что и класс, исключая:

- конструктор без параметров;
- инициализаторы полей;
- финализатор;
- виртуальные или защищенные члены.

Структура подходит там, где желательно иметь семантику типа значения. Хорошими примерами могут служить числовые типы, для которых более естественным способом присваивания является копирование значения, а не ссылки. Поскольку структура представляет собой тип значения, каждый экземпляр не требует создания объекта в куче; это дает ощутимую экономию при создании большого количества экземпляров типа. Например, создание массива с элементами типа значения требует только одного выделения памяти в куче.

## Семантика конструирования структуры

Семантика конструирования структуры выглядит следующим образом.

- Неявно существует конструктор без параметров, который невозможно переопределить. Он выполняет побитовое обнуление полей структуры.
- При определении конструктора (с параметрами) структуры каждому полю должно быть явно присвоено значение.

(Инициализаторы полей в структуре не предусмотрены.) Ниже показан пример объявления и вызова конструкторов структуры:

```
public struct Point
{
  int x, y;
  public Point (int x, int y) { this.x = x; this.y = y; }
}
...
Point p1 = new Point ();       // p1.x и p1.y будут равны 0
Point p2 = new Point (1, 1);   // p1.x и p1.y будут равны 1
```

Следующий пример приведет к возникновению трех ошибок на этапе компиляции:

```
public struct Point
{
  int x = 1;              // Не допускается: нельзя инициализировать поле
  int y;
  public Point() {}       // Не допускается: нельзя иметь конструктор
                          // без параметров
  public Point (int x) {this.x = x;}     // Не допускается: поле y должно
                                         // быть присвоено
}
```

Изменение `struct` на `class` сделает пример допустимым.

# Модификаторы доступа

Для содействия инкапсуляции тип или член типа может ограничивать свою *доступность* другим типам и сборкам за счет добавления к объявлению одного из пяти *модификаторов доступа*, которые описаны далее.

`public`

Полная доступность. Это неявная доступность для членов перечисления либо интерфейса.

`internal`

Доступность только внутри содержащей сборки или в дружественных сборках. Это стандартная доступность для невложенных типов.

`private`

Доступность только внутри содержащего типа. Это стандартная доступность для членов класса или структуры.

`protected`

Доступность только внутри содержащего типа или в его подклассах.

`protected internal`

*Объединение* доступностей `protected` и `internal`. Эрик Липперт объясняет это следующим образом: по умолчанию все является насколько возможно закрытым, и каждый модификатор делает что-то *более доступным*. Таким образом, применение `protected internal` к чему-либо делает его более доступным двумя путями.

В среде CLR имеется концепция *пересечения* доступностей `protected` и `internal`, но в языке C# она не поддерживается.

## Примеры

Класс `Class2` доступен извне его сборки; `Class1` — нет:

```
class Class1 {}  // Class1 является internal (по умолчанию)
public class Class2 {}
```

Класс ClassB открывает поле x другим типам в той же сборке; ClassA — нет:

```
class ClassA { int x;          }  // x является private (по умолчанию)
class ClassB { internal int x; }
```

Функции внутри Subclass могут вызывать Bar, но не Foo:

```
class BaseClass
{
  void Foo()           {}      // Foo является private (по умолчанию)
  protected void Bar() {}
}

class Subclass : BaseClass
{
  void Test1() { Foo(); }      // Ошибка - доступ к Foo невозможен
  void Test2() { Bar(); }      // Нормально
}
```

## Дружественные сборки

В более сложных сценариях члены internal можно открывать другим *дружественным* сборкам, добавляя атрибут сборки System.Runtime.CompilerServices.InternalsVisibleTo, в котором указано имя дружественной сборки:

```
[assembly: InternalsVisibleTo ("Friend")]
```

Если дружественная сборка имеет строгое имя (глава 18), тогда потребуется указать ее *полный* 160-байтовый открытый ключ:

```
[assembly: InternalsVisibleTo ("StrongFriend, PublicKey=0024f000048c...")]
```

Извлечь полный открытый ключ из строго именованной сборки можно с помощью запроса LINQ (более детально LINQ рассматривается в главе 8):

```
string key = string.Join ("",
  Assembly.GetExecutingAssembly().GetName().GetPublicKey()
    .Select (b => b.ToString ("x2")));
```

В сопровождающем книгу примере для LINQPad предлагается выбрать сборку и затем скопировать полный открытый ключ сборки в буфер обмена.

## Установление верхнего предела доступности

Тип устанавливает верхний предел доступности объявленных в нем членов. Наиболее распространенным примером такого установления является ситуация, когда есть тип internal с членами public. Например:

```
class C { public void Foo() {} }
```

Стандартная доступность internal класса C устанавливает верхний предел доступности метода Foo, по существу делая Foo внутренним. Распространенная причина пометки Foo как public связана с облегчением рефакторинга, если позже будет решено изменить доступность класса C на public.

## Ограничения, накладываемые на модификаторы доступа

При переопределении метода из базового класса доступность должна быть идентичной доступности переопределяемого метода. Например:

```
class BaseClass             { protected virtual  void Foo() {} }
class Subclass1 : BaseClass { protected override void Foo() {} }  // Нормально
class Subclass2 : BaseClass { public    override void Foo() {} }  // Ошибка
```

(Исключением является случай переопределения метода `protected internal` в другой сборке, при котором переопределяемый метод должен быть просто `protected`.)

Компилятор предотвращает несогласованное использование модификаторов доступа. Например, подкласс может иметь меньшую доступность, чем базовый класс, но не бо́льшую:

```
internal class A {}
public class B : A {}  // Ошибка
```

# Интерфейсы

Интерфейс похож на класс, но обеспечивает для своих членов только спецификацию, а не реализацию. Интерфейс обладает следующими особенностями.

- Все члены интерфейса являются *неявно абстрактными*. В противоположность этому класс может предоставлять как абстрактные члены, так и конкретные члены с реализациями.
- Класс (или структура) может реализовывать *несколько* интерфейсов. В противоположность этому класс может быть унаследован только от *одного* класса, а структура вообще не поддерживает наследование (за исключением того, что она порождена от `System.ValueType`).

Объявление интерфейса похоже на объявление класса, но никакой реализации для его членов не предоставляется, т.к. все члены интерфейса неявно абстрактные. Такие члены будут реализованы классами и структурами, которые реализуют данный интерфейс. Интерфейс может содержать только методы, свойства, события и индексаторы, что совершенно неслучайно в точности соответствует членам класса, которые могут быть абстрактными.

Ниже показано определение интерфейса `IEnumerator` из пространства имен `System.Collections`:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Члены интерфейса всегда неявно являются `public`, и для них нельзя объявлять какие-либо модификаторы доступа. Реализация интерфейса означает предоставление реализации `public` для всех его членов:

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext() => count-- > 0;
  public object Current => count;
  public void Reset() { throw new NotSupportedException(); }
}
```

Объект можно неявно приводить к любому интерфейсу, который он реализует. Например:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
  Console.Write (e.Current);  // 109876543210
```

Несмотря на то что `Countdown` является внутренним классом, его члены, которые реализуют интерфейс `IEnumerator`, могут вызываться открытым образом за счет приведения экземпляра `Countdown` к `IEnumerator`. Скажем, если какой-то открытый тип в той же сборке определяет метод, как показано ниже:

```
public static class Util
{
  public static object GetCountDown() => new CountDown();
}
```

то в вызывающем коде внутри другой сборки можно поступать так:

```
IEnumerator e = (IEnumerator) Util.GetCountDown();
e.MoveNext();
```

Если бы сам интерфейс `IEnumerator` был определен как `internal`, то подобное оказалось бы невозможным.

## Расширение интерфейса

Интерфейсы могут быть производными от других интерфейсов. Например:

```
public interface IUndoable             { void Undo(); }
public interface IRedoable : IUndoable { void Redo(); }
```

Интерфейс `IRedoable` "наследует" все члены интерфейса `IUndoable`. Другими словами, типы, которые реализуют `IRedoable`, должны также реализовывать члены `IUndoable`.

## Явная реализация членов интерфейса

Реализация множества интерфейсов иногда может приводить к конфликту между сигнатурами членов. Разрешать такие конфликты можно за счет *явной реализации* члена интерфейса. Рассмотрим следующий пример:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2
{
  public void Foo()
  {
    Console.WriteLine ("Widget's implementation of I1.Foo");
  }

  int I2.Foo()
  {
    Console.WriteLine ("Widget's implementation of I2.Foo");
    return 42;
  }
}
```

Поскольку интерфейсы I1 и I2 имеют методы Foo с конфликтующими сигнатурами, метод Foo интерфейса I2 в классе Widget реализуется явно. Это позволяет двум методам сосуществовать в рамках одного класса. Единственный способ вызова явно реализованного метода предусматривает приведение к его интерфейсу:

```
Widget w = new Widget();
w.Foo();            // Реализация I1.Foo из Widget
((I1)w).Foo();      // Реализация I1.Foo из Widget
((I2)w).Foo();      // Реализация I2.Foo из Widget
```

Другой причиной явной реализации членов интерфейса может быть необходимость сокрытия членов, которые являются узкоспециализированными и нарушающими нормальный сценарий применения типа. Скажем, тип, который реализует ISerializable, обычно будет избегать демонстрации членов ISerializable, если только не осуществляется явное приведение к упомянутому интерфейсу.

## Реализация виртуальных членов интерфейса

Неявно реализованный член интерфейса по умолчанию является запечатанным. Чтобы его можно было переопределить, он должен быть помечен в базовом классе как virtual или abstract. Например:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  public virtual void Undo() => Console.WriteLine ("TextBox.Undo");
}
public class RichTextBox : TextBox
{
  public override void Undo() => Console.WriteLine ("RichTextBox.Undo");
}
```

Обращение к этому члену интерфейса либо через базовый класс, либо интерфейс приводит к вызову его реализации из подкласса:

```
RichTextBox r = new RichTextBox();
r.Undo();                 // RichTextBox.Undo
((IUndoable)r).Undo();    // RichTextBox.Undo
((TextBox)r).Undo();      // RichTextBox.Undo
```

Явно реализованный член интерфейса не может быть помечен как virtual, равно как и не может быть переопределен обычным образом. Однако он может быть *повторно реализован*.

## Повторная реализация члена интерфейса в подклассе

Подкласс может повторно реализовать любой член интерфейса, который уже реализован базовым классом. Повторная реализация перехватывает реализацию члена (при вызове через интерфейс) и работает вне зависимости от того, является ли член виртуальным в базовом классе. Повторная реализация также работает в ситуации, когда член реализован неявно или явно – хотя, как будет продемонстрировано, в последнем случае она работает лучше.

В показанном ниже примере класс TextBox явно реализует IUndoable.Undo, а потому данный метод не может быть помечен как virtual. Чтобы "переопределить" его, класс RichTextBox должен повторно реализовать метод Undo интерфейса IUndoable:

```
public interface IUndoable { void Undo(); }
public class TextBox : IUndoable
{
  void IUndoable.Undo() => Console.WriteLine ("TextBox.Undo");
}
public class RichTextBox : TextBox, IUndoable
{
  public void Undo() => Console.WriteLine ("RichTextBox.Undo");
}
```

Обращение к повторно реализованному методу через интерфейс приводит к вызову его реализации из подкласса:

```
RichTextBox r = new RichTextBox();
r.Undo();                // RichTextBox.Undo  Случай 1
((IUndoable)r).Undo();   // RichTextBox.Undo  Случай 2
```

При том же самом определении RichTextBox предположим, что TextBox реализует метод Undo *неявно*:

```
public class TextBox : IUndoable
{
  public void Undo() => Console.WriteLine ("TextBox.Undo");
}
```

Это дает еще один способ вызова метода Undo, который "нарушает" систему, как показано в случае 3:

```
RichTextBox r = new RichTextBox();
r.Undo();                // RichTextBox.Undo  Случай 1
((IUndoable)r).Undo();   // RichTextBox.Undo  Случай 2
((TextBox)r).Undo();     // TextBox.Undo      Случай 3
```

Случай 3 демонстрирует тот факт, что перехват повторной реализации результативен, только когда член вызывается через интерфейс, а не через базовый класс. Обычно подобное нежелательно, т.к. может означать несогласованную семантику. Это делает повторную реализацию наиболее подходящей в качестве стратегии для переопределения *явно* реализованных членов интерфейса.

### Альтернативы повторной реализации членов интерфейса

Даже при явной реализации членов повторная реализация проблематична по следующим причинам.

- Подкласс не имеет возможности вызывать метод базового класса.
- Автор базового класса мог не предполагать, что метод будет повторно реализован, и потому не учел потенциальные последствия.

Повторная реализация может оказаться хорошим последним средством в ситуации, когда создание подклассов не предвиделось. Тем не менее, лучше проектировать базовый класс так, чтобы потребность в повторной реализации никогда не возникала. Этого можно достичь двумя путями:

- в случае неявной реализации члена пометьте его как virtual, если подобное возможно;
- в случае явной реализации члена используйте следующий шаблон, если предполагается, что в подклассах может понадобиться переопределение любой логики:

```
public class TextBox : IUndoable
{
  void IUndoable.Undo() => Undo();  // Вызывает метод, определенный ниже
  protected virtual void Undo() => Console.WriteLine ("TextBox.Undo");
}
public class RichTextBox : TextBox
{
  protected override void Undo() => Console.WriteLine("RichTextBox.Undo");
}
```

Если создание подклассов не предвидится, тогда класс можно пометить как `sealed`, чтобы предотвратить повторную реализацию членов интерфейса.

## Интерфейсы и упаковка

Преобразование структуры в интерфейс приводит к упаковке. Обращение к неявно реализованному члену структуры упаковку не вызывает:

```
interface  I { void Foo();          }
struct S : I { public void Foo() {} }

...
S s = new S();
s.Foo();   // Упаковка не происходит

I i = s;   // Упаковка происходит во время приведения к интерфейсу
i.Foo();
```

### Написание кода класса или кода интерфейса

Запомните в качестве руководства:

- применяйте классы и подклассы для типов, которые естественным образом разделяют некоторую реализацию;
- используйте интерфейсы для типов, которые имеют независимые реализации.

Рассмотрим следующие классы:

```
abstract class Animal {}
abstract class Bird           : Animal {}
abstract class Insect         : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore      : Animal {}

// Конкретные классы:

class Ostrich : Bird {}
class Eagle   : Bird, FlyingCreature, Carnivore {}   // Не допускается
class Bee     : Insect, FlyingCreature {}            // Не допускается
class Flea    : Insect, Carnivore {}                 // Не допускается
```

Код классов `Eagle`, `Bee` и `Flea` не скомпилируется, потому что наследование от нескольких классов запрещено. Чтобы решить такую проблему, мы должны преобразовать некоторые типы в интерфейсы. Здесь и возникает вопрос: какие именно типы? Следуя главному правилу, мы можем сказать, что насекомые (`Insect`) разделяют реализацию, и птицы (`Bird`) разделяют реализацию, а потому они остаются классами. В противоположность им летающие существа (`FlyingCreature`) имеют независимые механизмы для полета, а плотоядные животные (`Carnivore`) под-

держивают независимые линии поведения при поедании, так что мы можем преобразовать FlyingCreature и Carnivore в интерфейсы:

```
interface IFlyingCreature {}
interface ICarnivore      {}
```

В типичном сценарии классы Bird и Insect могут соответствовать элементу управления Windows и веб-элементу управления, а FlyingCreature и Carnivore — интерфейсам IPrintable и IUndoable.

# Перечисления

Перечисление — это специальный тип значения, который позволяет указывать группу именованных числовых констант. Например:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Такое перечисление можно применять следующим образом:

```
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);   // true
```

Каждый член перечисления имеет лежащее в его основе целочисленное значение. По умолчанию:

- лежащие в основе значения относятся к типу int;
- членам перечисления присваиваются константы 0, 1, 2... (в порядке их объявления).

Можно указать другой целочисленный тип:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

Для каждого члена перечисления можно также указывать явные лежащие в основе значения:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```

Компилятор также позволяет явно присваивать значения *определенным* членам перечисления. Члены, которым не были присвоены значения, получают значения на основе инкрементирования последнего явно указанного значения. Предыдущий пример эквивалентен следующему коду:

```
public enum BorderSide : byte
 { Left=1, Right, Top=10, Bottom }
```

## Преобразования перечислений

Экземпляр перечисления может быть преобразован в и из лежащего в основе целочисленного значения с помощью явного приведения:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftOrRight = (int) side <= 2;
```

Можно также явно приводить один тип перечисления к другому. Предположим, что определение HorizontalAlignment выглядит следующим образом:

```
public enum HorizontalAlignment
{
```

```
  Left = BorderSide.Left,
  Right = BorderSide.Right,
  Center
}
```

При трансляции между типами перечислений используются лежащие в их основе целочисленные значения:

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;
// То же самое, что и:
HorizontalAlignment h = (HorizontalAlignment) (int) BorderSide.Right;
```

Числовой литерал 0 в выражении enum трактуется компилятором особым образом и явного приведения не требует:

```
BorderSide b = 0;  // Приведение не требуется
if (b == 0) ...
```

Существуют две причины для специальной трактовки значения 0:

- первый член перечисления часто применяется как "стандартное" значение;
- для типов *комбинированных перечислений* значение 0 означает "отсутствие флагов".

## Перечисления флагов

Члены перечислений можно комбинировать. Чтобы предотвратить неоднозначности, члены комбинируемого перечисления требуют явного присваивания значений, обычно являющихся степенью двойки. Например:

```
[Flags]
public enum BorderSides { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

При работе со значениями комбинированного перечисления используются побитовые операции, такие как | и &. Они имеют дело с лежащими в основе целыми значениями:

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;

if ((leftRight & BorderSides.Left) != 0)
  Console.WriteLine ("Includes Left");      // Включает Left

string formatted = leftRight.ToString();    // "Left, Right"

BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine (s == leftRight);         // True

s ^= BorderSides.Right;  // Переключает BorderSides.Right
Console.WriteLine (s);   // Left
```

По соглашению к типу перечисления всегда должен применяться атрибут Flags, когда члены перечисления являются комбинируемыми. Если объявить такое перечисление без атрибута Flags, то комбинировать его члены по-прежнему можно будет, но вызов ToString на экземпляре перечисления будет выдавать число, а не последовательность имен.

По соглашению типу комбинируемого перечисления назначается имя во множественном, а не единственном числе.

Для удобства члены комбинаций могут быть помещены в само объявление перечисления:

```
[Flags]
public enum BorderSides
{
  None=0,
  Left=1, Right=2, Top=4, Bottom=8,
  LeftRight = Left | Right,
  TopBottom = Top  | Bottom,
  All       = LeftRight | TopBottom
}
```

## Операции над перечислениями

Ниже указаны операции, которые могут работать с перечислениями:

```
=   ==   !=   <   >   <=   >=   +   -   ^   &   |   ~
+=   -=   ++  --   sizeof
```

Побитовые, арифметические и операции сравнения возвращают результат обработки лежащих в основе целочисленных значений. Сложение разрешено для перечисления и целочисленного типа, но не для двух перечислений.

## Проблемы безопасности типов

Рассмотрим следующее перечисление:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Поскольку тип перечисления может быть приведен к лежащему в основе целому типу и наоборот, фактическое значение может выходить за пределы допустимых границ для членов перечисления. Например:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);  // 12345
```

Побитовые и арифметические операции могут аналогично давать в результате недопустимые значения:

```
BorderSide b = BorderSide.Bottom;
b++;  // Ошибки не возникают
```

Недопустимый экземпляр `BorderSide` может нарушить работу следующего кода:

```
void Draw (BorderSide side)
{
  if      (side == BorderSide.Left)  {...}
  else if (side == BorderSide.Right) {...}
  else if (side == BorderSide.Top)   {...}
  else                               {...}  // Предполагается BorderSide.Bottom
}
```

Одно из решений предусматривает добавление дополнительной конструкции `else`:

```
...
else if (side == BorderSide.Bottom) ...
else throw new ArgumentException ("Invalid BorderSide: " + side, "side");
                                  // Недопустимое значение BorderSide
```

Еще один обходной прием заключается в явной проверке значения перечисления на предмет допустимости. Такую работу выполняет статический метод `Enum.IsDefined`:

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine (Enum.IsDefined (typeof (BorderSide), side));  // False
```

К сожалению, метод `Enum.IsDefined` не работает с перечислениями флагов. Однако показанный далее вспомогательный метод (трюк, зависящий от поведения `Enum.ToString`) возвращает `true`, если заданное перечисление флагов является допустимым:

```
static bool IsFlagDefined (Enum e)
{
  decimal d;
  return !decimal.TryParse(e.ToString(), out d);
}

[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
static void Main()
{
  for (int i = 0; i <= 16; i++)
  {
    BorderSides side = (BorderSides)i;
    Console.WriteLine (IsFlagDefined (side) + " " + side);
  }
}
```

# Вложенные типы

*Вложенный тип* объявляется внутри области видимости другого типа. Например:

```
public class TopLevel
{
  public class Nested { }               // Вложенный класс
  public enum Color { Red, Blue, Tan }  // Вложенное перечисление
}
```

Вложенный тип обладает следующими характеристиками.

- Он может получать доступ к закрытым членам включающего типа и ко всему остальному, к чему включающий тип имеет доступ.
- Он может быть объявлен с полным диапазоном модификаторов доступа, а не только `public` и `internal`.
- Стандартной доступностью вложенного типа является `private`, а не `internal`.
- Доступ к вложенному типу извне требует указания имени включающего типа (как при обращении к статическим членам).

Например, для доступа к члену `Color.Red` извне класса `TopLevel` необходимо записать такой код:

```
TopLevel.Color color = TopLevel.Color.Red;
```

Вложение в класс или структуру допускают все типы (классы, структуры, интерфейсы, делегаты и перечисления).

Ниже приведен пример обращения к закрытому члену типа из вложенного типа:

```
public class TopLevel
{
  static int x;
```

```
  class Nested
  {
    static void Foo() { Console.WriteLine (TopLevel.x); }
  }
}
```

А вот пример использования модификатора доступа `protected` с вложенным типом:

```
public class TopLevel
{
  protected class Nested { }
}
public class SubTopLevel : TopLevel
{
  static void Foo() { new TopLevel.Nested(); }
}
```

Далее показан пример ссылки на вложенный тип извне включающего типа:

```
public class TopLevel
{
  public class Nested { }
}
class Test
{
  TopLevel.Nested n;
}
```

Вложенные типы интенсивно применяются самим компилятором, когда он генерирует закрытые классы, которые хранят состояние для таких конструкций, как итераторы и анонимные методы.

Если единственной причиной для использования вложенного типа является желание избежать загромождения пространства имен слишком большим числом типов, тогда рассмотрите возможность применения взамен вложенного пространства имен. Вложенный тип должен использоваться из-за его более строгих ограничений контроля доступа или же когда вложенному классу нужен доступ к закрытым членам включающего класса.

# Обобщения

В C# имеются два отдельных механизма для написания кода, многократно применяемого различными типами: *наследование* и *обобщения*. В то время как наследование выражает повторное использование с помощью базового типа, обобщения делают это посредством "шаблона", который содержит "типы-заполнители". В сравнении с наследованием обобщения могут *увеличивать безопасность типов*, а также *сокращать количество приведений и упаковок*.

Обобщения C# и шаблоны C++ — похожие концепции, но работают они по-разному. Разница объясняется в разделе "Сравнение обобщений C# и шаблонов C++" в конце настоящей главы.

## Обобщенные типы

Обобщенный тип объявляет *параметры типа* – типы-заполнители, предназначенные для заполнения потребителем обобщенного типа, который предоставляет *аргументы типа*. Ниже показан обобщенный тип `Stack<T>`, предназначенный для реализации стека экземпляров типа `T`. В `Stack<T>` объявлен единственный параметр типа `T`:

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj) => data[position++] = obj;
  public T Pop()           => data[--position];
}
```

Вот как можно применять `Stack<T>`:

```
var stack = new Stack<int>();
stack.Push (5);
stack.Push (10);
int x = stack.Pop();  // x имеет значение 10
int y = stack.Pop();  // y имеет значение 5
```

Класс `Stack<int>` заполняет параметр типа `T` аргументом типа `int`, неявно создавая тип на лету (синтез происходит во время выполнения). Однако попытка помещения в стек типа `Stack<int>` строки приведет к ошибке на этапе компиляции. Фактически `Stack<int>` имеет показанное ниже определение (подстановки выделены полужирным, и во избежание путаницы вместо имени класса указано `###`):

```
public class ###
{
  int position;
  int[] data;
  public void Push (int obj) => data[position++] = obj;
  public int Pop()           => data[--position];
}
```

Формально мы говорим, что `Stack<T>` – это *открытый (open) тип*, а `Stack<int>` – *закрытый (closed) тип*. Во время выполнения все экземпляры обобщенных типов закрываются – с заполнением их типов-заполнителей. Это значит, что показанный ниже оператор является недопустимым:

```
var stack = new Stack<T>();  // Не допускается: что собой представляет T?
```

если только он не находится внутри класса или метода, который сам определяет `T` как параметр типа:

```
public class Stack<T>
{
  ...
  public Stack<T> Clone()
  {
    Stack<T> clone = new Stack<T>();  // Допускается
    ...
  }
}
```

## Для чего предназначены обобщения

Обобщения предназначены для записи кода, который может многократно использоваться различными типами. Предположим, что нам нужен стек целочисленных значений, но мы не располагаем обобщенными типами. Одно из решений предусматривает жесткое кодирование отдельной версии класса для каждого требуемого типа элементов (например, `IntStack`, `StringStack` и т.д.). Очевидно, что такой подход приведет к дублированию значительного объема кода. Другое решение заключается в написании стека, который обобщается за счет применения `object` в качестве типа элементов:

```
public class ObjectStack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj) => data[position++] = obj;
  public object Pop()           => data[--position];
}
```

Тем не менее, класс `ObjectStack` не будет работать настолько же эффективно, как жестко закодированный класс `IntStack`, предназначенный для сохранения в стеке целочисленных значений. В частности, `ObjectStack` будет требовать упаковки и приведения вниз, которые не могут быть проверены на этапе компиляции:

```
// Предположим, что мы просто хотим сохранять целочисленные значения:
ObjectStack stack = new ObjectStack();

stack.Push ("s");              // Некорректный тип, но ошибка не возникает!
int i = (int)stack.Pop();      // Приведение вниз - ошибка времени выполнения
```

Нас интересует универсальная реализация стека, работающая со всеми типами элементов, а также возможность ее легкой специализации для конкретного типа элементов в целях повышения безопасности типов и сокращения приведений и упаковок. Именно это обеспечивают обобщения, позволяя параметризировать тип элементов. Тип `Stack<T>` обладает преимуществами и `ObjectStack`, и `IntStack`. Подобно `ObjectStack` класс `Stack<T>` написан один раз для *универсальной* работы со всеми типами. Как и `IntStack`, класс `Stack<T>` *специализируется* для конкретного типа — его элегантность в том, что таким типом является `T`, который можно подставлять на лету.

Класс `ObjectStack` функционально эквивалентен `Stack<object>`.

## Обобщенные методы

Обобщенный метод объявляет параметры типа внутри сигнатуры метода.

С помощью обобщенных методов многие фундаментальные алгоритмы могут быть реализованы единственным универсальным способом. Ниже показан обобщенный метод, который меняет местами содержимое двух переменных любого типа `T`:

```
static void Swap<T> (ref T a, ref T b)
{
  T temp = a;
  a = b;
  b = temp;
}
```

Метод Swap<T> можно использовать следующим образом:

```
int x = 5;
int y = 10;
Swap (ref x, ref y);
```

Как правило, предоставлять аргументы типа обобщенному методу нет нужды, поскольку компилятор способен неявно вывести тип. Если имеется неоднозначность, то обобщенные методы могут быть вызваны с аргументами типа:

```
Swap<int> (ref x, ref y);
```

Внутри обобщенного *типа* метод не классифицируется как обобщенный, если только он не *вводит* параметры типа (посредством синтаксиса с угловыми скобками). Метод Pop в нашем обобщенном стеке просто задействует существующий параметр типа T и не трактуется как обобщенный.

Методы и типы — единственные конструкции, в которых могут вводиться параметры типа. Свойства, индексаторы, события, поля, конструкторы, операции и т.д. не могут объявлять параметры типа, хотя могут пользоваться любыми параметрами типа, которые уже объявлены во включающем типе. В примере с обобщенным стеком можно было бы написать индексатор, который возвращает обобщенный элемент:

```
public T this [int index] => data [index];
```

Аналогично конструкторы также могут пользоваться существующими параметрами типа, но не *вводить* их:

```
public Stack<T>() { }  // Не допускается
```

## Объявление параметров типа

Параметры типа могут вводиться в объявлениях классов, структур, интерфейсов, делегатов (рассматриваются в главе 4) и методов. Другие конструкции, такие как свойства, не могут *вводить* параметры типа, но могут их *потреблять*. Например, свойство Value использует T:

```
public struct Nullable<T>
{
  public T Value { get; }
}
```

Обобщенный тип или метод может иметь несколько параметров. Например:

```
class Dictionary<TKey, TValue> {...}
```

Вот как создать его экземпляр:

```
Dictionary<int,string> myDic = new Dictionary<int,string>();
```

Или так:

```
var myDic = new Dictionary<int,string>();
```

Имена обобщенных типов и методов могут быть перегружены при условии, что количество параметров типа у них отличается. Например, показанные ниже три имени типа не конфликтуют друг с другом:

```
class A        {}
class A<T>     {}
class A<T1,T2> {}
```

По соглашению обобщенные типы и методы с *единственным* параметром типа обычно именуют его как `T`, если назначение параметра очевидно. В случае *нескольких* параметров типа каждый такой параметр имеет более описательное имя (с префиксом `T`).

## Операция `typeof` и несвязанные обобщенные типы

Во время выполнения открытых обобщенных типов не существует: они закрываются на этапе компиляции. Однако во время выполнения возможно существование *несвязанного* (unbound) обобщенного типа — исключительно как объекта `Type`. Единственным способом указания несвязанного обобщенного типа в C# является применение операции `typeof`:

```
class A<T> {}
class A<T1,T2> {}
...
Type a1 = typeof (A<>);   // Несвязанный тип (обратите внимание
                          // на отсутствие аргументов типа)
Type a2 = typeof (A<,>);  // При указании нескольких аргументов типа
                          // используются запятые
```

Открытые обобщенные типы применяются в сочетании с Reflection API (глава 19). Операцию `typeof` можно также использовать для указания закрытого типа:

```
Type a3 = typeof (A<int,int>);
```

или открытого типа (который закроется во время выполнения):

```
class B<T> { void X() { Type t = typeof (T); } }
```

## Стандартное значение обобщенного параметра типа

Ключевое слово `default` может применяться для получения стандартного значения обобщенного параметра типа. Стандартным значением для ссылочного типа является `null`, а для типа значения — результат побитового обнуления полей в этом типе:

```
static void Zap<T> (T[] array)
{
  for (int i = 0; i < array.Length; i++)
    array[i] = default(T);
}
```

## Ограничения обобщений

По умолчанию параметр типа может быть замещен любым типом. Чтобы затребовать более специфичные аргументы типа, к параметру типа можно применить *ограничения*. Существуют шесть видов ограничений:

```
where T : базовый-класс  // Ограничение базового класса
where T : интерфейс      // Ограничение интерфейса
where T : class          // Ограничение ссылочного типа
where T : struct         // Ограничение типа значения
                         //  (исключает типы, допускающие null)
where T : new()          // Ограничение конструктора без параметров
where U : T              // Неприкрытое ограничение типа
```

В следующем примере GenericClass<T,U> требует, чтобы тип T был производным от класса SomeClass (или идентичен ему) и реализовывал интерфейс Interface1, а тип U предоставлял конструктор без параметров:

```
class     SomeClass {}
interface Interface1 {}

class GenericClass<T,U> where T : SomeClass, Interface1
                        where U : new()
{...}
```

Ограничения могут использоваться везде, где определены параметры типа, как в методах, так и в определениях типов.

*Ограничение базового класса* указывает, что параметр типа должен быть подклассом заданного класса (или совпадать с ним); *ограничение интерфейса* указывает, что параметр типа должен реализовывать этот интерфейс. Такие ограничения позволяют экземплярам параметра типа быть неявно преобразуемыми в этот класс или интерфейс. Например, пусть необходимо написать обобщенный метод Max, который возвращает большее из двух значений. Мы можем задействовать обобщенный интерфейс IComparable<T>, определенный в .NET Framework:

```
public interface IComparable<T>  // Упрощенная версия интерфейса
{
  int CompareTo (T other);
}
```

Метод CompareTo возвращает положительное число, если this больше other. Применяя данный интерфейс в качестве ограничения, мы можем написать метод Max следующим образом (чтобы не отвлекать внимание, проверка на null опущена):

```
static T Max <T> (T a, T b) where T : IComparable<T>
{
  return a.CompareTo (b) > 0 ? a : b;
}
```

Метод Max может принимать аргументы любого типа, реализующего интерфейс IComparable<T> (что включает большинство встроенных типов, таких как int и string):

```
int z = Max (5, 10);                // 10
string last = Max ("ant", "zoo");   // zoo
```

*Ограничение class* и *ограничение struct* указывают, что T должен быть ссылочным типом или типом значения (не допускающим null). Хорошим примером ограничения struct является структура System.Nullable<T> (мы обсудим этот тип в разделе "Типы, допускающие значение null" главы 4):

```
struct Nullable<T> where T : struct {...}
```

*Ограничение конструктора без параметров* требует, чтобы тип T имел открытый конструктор без параметров и позволял вызывать операцию new на T:

```
static void Initialize<T> (T[] array) where T : new()
{
  for (int i = 0; i < array.Length; i++)
    array[i] = new T();
}
```

*Неприкрытое ограничение типа* требует, чтобы один параметр типа был производным от другого параметра типа (или совпадал с ним). В следующем примере метод

FilteredStack возвращает другой экземпляр Stack, содержащий только подмножество элементов, в которых параметр типа U является параметром типа T:

```
class Stack<T>
{
  Stack<U> FilteredStack<U>() where U : T {...}
}
```

## Создание подклассов для обобщенных типов

Подклассы для обобщенного класса можно создавать точно так же, как в случае необобщенного класса. Подкласс может оставлять параметры типа базового класса открытыми, как показано в следующем примере:

```
class Stack<T>                    {...}
class SpecialStack<T> : Stack<T> {...}
```

Либо же подкласс может закрыть параметры обобщенного типа посредством конкретного типа:

```
class IntStack : Stack<int>  {...}
```

Подкласс может также вводить новые аргументы типа:

```
class List<T>                       {...}
class KeyedList<T,TKey> : List<T> {...}
```

Формально *все* аргументы типа в подтипе являются новыми: можно сказать, что подтип закрывает и затем повторно открывает аргументы базового типа. Это значит, что подкласс может назначать аргументам типа новые (и потенциально более осмысленные) имена, когда повторно их открывает:

```
class List<T> {...}
class KeyedList<TElement,TKey> : List<TElement> {...}
```

## Самоссылающиеся объявления обобщений

Тип может указывать *самого себя* в качестве конкретного типа при закрытии аргумента типа:

```
public interface IEquatable<T> { bool Equals (T obj); }

public class Balloon : IEquatable<Balloon>
{
  public string Color { get; set; }
  public int CC { get; set; }

  public bool Equals (Balloon b)
  {
    if (b == null) return false;
    return b.Color == Color && b.CC == CC;
  }
}
```

Следующий код также допустим:

```
class Foo<T> where T : IComparable<T> { ... }
class Bar<T> where T : Bar<T> { ... }
```

## Статические данные

Статические данные являются уникальными для каждого закрытого типа:

```
class Bob<T> { public static int Count; }

class Test
{
  static void Main()
  {
    Console.WriteLine (++Bob<int>.Count);        // 1
    Console.WriteLine (++Bob<int>.Count);        // 2
    Console.WriteLine (++Bob<string>.Count);     // 1
    Console.WriteLine (++Bob<object>.Count);     // 1
  }
}
```

## Параметры типа и преобразования

Операция приведения в C# может выполнять преобразования нескольких видов, включая:

- числовое преобразование;
- ссылочное преобразование;
- упаковывающее/распаковывающее преобразование;
- специальное преобразование (через перегрузку операций; см. главу 4).

Решение о том, какой вид преобразования произойдет, принимается *на этапе компиляции*, базируясь на известных типах операндов. Это создает интересный сценарий с параметрами обобщенного типа, т.к. точные типы операндов на этапе компиляции не известны. Если возникает неоднозначность, тогда компилятор генерирует сообщение об ошибке.

Наиболее распространенный сценарий связан с выполнением ссылочного преобразования:

```
StringBuilder Foo<T> (T arg)
{
  if (arg is StringBuilder)
    return (StringBuilder) arg;  // Не скомпилируется
  ...
}
```

Без знания действительного типа `T` компилятор предполагает, что вы намереваетесь выполнить *специальное преобразование*. Простейшим решением будет использование взамен операции `as`, которая не дает неоднозначности, т.к. не позволяет осуществлять специальные преобразования:

```
StringBuilder Foo<T> (T arg)
{
  StringBuilder sb = arg as StringBuilder;
  if (sb != null) return sb;
  ...
}
```

Более общее решение предусматривает приведение сначала к `object`. Такой подход работает, поскольку предполагается, что преобразования в/из `object` должны

быть не специальными, а ссылочными или упаковывающими/распаковывающими. В данном случае StringBuilder является ссылочным типом, поэтому должно происходить ссылочное преобразование:

```
return (StringBuilder) (object) arg;
```

Распаковывающие преобразования также способны привносить неоднозначность. Показанное ниже преобразование может быть распаковывающим, числовым или специальным:

```
int Foo<T> (T x) => (int) x;  // Ошибка на этапе компиляции
```

И снова решение заключается в том, чтобы сначала выполнить приведение к object, а затем к int (которое в данном случае однозначно сигнализирует о распаковывающем преобразовании):

```
int Foo<T> (T x) => (int) (object) x;
```

## Ковариантность

Исходя из предположения, что тип A допускает преобразование в B, тип X имеет ковариантный параметр типа, если X<A> поддается преобразованию в X<B>.

Согласно понятию вариантности в языке C# формулировка "поддается преобразованию" означает возможность преобразования через *неявное ссылочное преобразование* – такое как *A является подклассом B* или *A реализует B*. Сюда не входят числовые преобразования, упаковывающие преобразования и специальные преобразования.

Например, тип IFoo<T> имеет ковариантный тип T, если справедливо следующее:

```
IFoo<string> s = ...;
IFoo<object> b = s;
```

Начиная с версии C# 4.0, интерфейсы допускают ковариантные параметры типа (как это делают делегаты – см. главу 4) , но обобщенные классы – нет. Массивы также разрешают ковариантность (массив A[] может быть преобразован в B[], если для A имеется ссылочное преобразование в B) и обсуждаются здесь для сравнения.

Ковариантность и контравариантность (или просто "вариантность") – сложные концепции. Мотивация, лежащая в основе введения и расширения вариантности в язык C#, заключалась в том, чтобы позволить обобщенным интерфейсам и обобщениям (в частности, определенным в .NET Framework, таким как IEnumerable<T>) работать *более предсказуемым образом*. Вы можете извлечь выгоду из этого, даже не понимая все детали ковариантности и контравариантности.

### Вариантность не является автоматической

Чтобы обеспечить статическую безопасность типов, параметры типа не являются автоматически вариантными. Рассмотрим приведенный ниже код:

```
class Animal {}
class Bear  : Animal {}
class Camel : Animal {}

public class Stack<T>  // Простая реализация стека
{
```

```
  int position;
  T[] data = new T[100];
  public void Push (T obj) => data[position++] = obj;
  public T Pop()           => data[--position];
}
```

Следующий код не скомпилируется:

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears;     // Ошибка на этапе компиляции
```

Это ограничение предотвращает возможность возникновения ошибки во время выполнения из-за такого кода:

```
animals.Push (new Camel());        // Попытка добавить объект Camel в bears
```

Тем не менее, отсутствие ковариантности может послужить препятствием повторному использованию. Предположим, что требуется написать код метода `Wash` (мыться) для стека `animals` (животные):

```
public class ZooCleaner
{
  public static void Wash (Stack<Animal> animals) {...}
}
```

Вызов метода `Wash` со стеком `bears` (медведи) приведет к генерации ошибки на этапе компиляции. Один из обходных путей предполагает переопределение метода `Wash` с ограничением:

```
class ZooCleaner
{
  public static void Wash<T> (Stack<T> animals) where T : Animal { ... }
}
```

Теперь метод `Wash` можно вызывать следующим образом:

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash (bears);
```

Другое решение состоит в том, чтобы обеспечить реализацию классом `Stack<T>` интерфейса с ковариантным параметром типа, как вскоре будет показано.

### Массивы

По историческим причинам типы массивов поддерживают ковариантность. Это значит, что массив `B[]` может быть приведен к `A[]`, если `B` является подклассом `A` (и оба они являются ссылочными типами). Например:

```
Bear[] bears = new Bear[3];
Animal[] animals = bears;  // Нормально
```

Недостаток такой возможности повторного использования заключается в том, что присваивание элементов может потерпеть неудачу во время выполнения:

```
animals[0] = new Camel();  // Ошибка во время выполнения
```

### Объявление ковариантного параметра типа

Начиная с версии C# 4.0, параметры типа в интерфейсах и делегатах могут быть объявлены как ковариантные путем их пометки с помощью модификатора `out`. Данный модификатор гарантирует, что в отличие от массивов ковариантные параметры типа будут полностью безопасными в отношении типов.

Мы можем проиллюстрировать сказанное на классе Stack, обеспечив реализацию им следующего интерфейса:

```
public interface IPoppable<out T> { T Pop(); }
```

Модификатор out для T указывает, что тип T применяется только в *выходных позициях* (например, в возвращаемых типах для методов). Модификатор out помечает параметр типа как *ковариантный* и разрешает написание такого кода:

```
var bears = new Stack<Bear>();
bears.Push (new Bear());
// bears реализует IPoppable<Bear>.
// Можно выполнить преобразование в IPoppable<Animal>:
IPoppable<Animal> animals = bears;  // Допустимо
Animal a = animals.Pop();
```

Преобразование bears в animals разрешено компилятором в силу того, что параметр типа является ковариантным. Преобразование безопасно в отношении типов, т.к. сценарий, от которого компилятор пытается уклониться (помещение объекта Camel в стек), возникнуть не может, поскольку нет способа передать Camel *в* интерфейс, где T может встречаться только в *выходных* позициях.

Ковариантность (и контравариантность) в интерфейсах — это то, что обычно *потребляется*: необходимость *написания* вариантных интерфейсов возникает реже.

Любопытно, что параметры метода, помеченные как out, не подходят для ковариантности из-за ограничения в среде CLR.

Возможность ковариантного приведения можно задействовать для решения описанной ранее проблемы повторного использования:

```
public class ZooCleaner
{
  public static void Wash (IPoppable<Animal> animals) { ... }
}
```

Интерфейсы IEnumerator<T> и IEnumerable<T>, описанные в главе 7, имеют ковариантный параметр типа T, что позволяет приводить IEnumerable<string>, например, к IEnumerable<object>.

Компилятор сгенерирует ошибку, если ковариантный параметр типа применяется во *входной* позиции (скажем, в параметре метода или в записываемом свойстве).

Ковариантность и контравариантность работают только для элементов со *ссылочными преобразованиями* — не *упаковывающими преобразованиями*. (Это применимо как к вариантности параметров типа, так и к вариантности массивов.) Таким образом, если имеется метод, который принимает параметр типа IPoppable<object>, то его можно вызывать с IPoppable<string>, но не с IPoppable<int>.

## Контравариантность

Как было показано ранее, если предположить, что A разрешает неявное ссылочное преобразование в B, то тип X имеет ковариантный параметр типа, когда X<A> допускает ссылочное преобразование в X<B>. *Контравариантность* присутствует в случае, если возможно преобразование в обратном направлении — из X<B> в X<A>. Это поддерживается, когда параметр типа встречается только во *входных* позициях, и обозначается с помощью модификатора in. Продолжая предыдущий пример, если класс Stack<T> реализует следующий интерфейс:

```
public interface IPushable<in T> { void Push (T obj); }
```

то вполне законно поступить так:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;  // Допустимо
bears.Push (new Bear());
```

Ни один из членов IPushable не содержит тип T в *выходной* позиции, а потому никаких проблем с приведением animals к bears не возникает (например, данный интерфейс не поддерживает метод Pop).

Класс Stack<T> может реализовывать оба интерфейса, IPushable<T> и IPoppable<T> — несмотря на то, что тип T в указанных двух интерфейсах имеет противоположные модификаторы вариантности! Причина в том, что вариантность должна использоваться через интерфейс, а не через класс; следовательно, перед выполнением вариантного преобразования его потребуется пропустить сквозь призму либо IPoppable, либо IPushable. В результате вы будете ограничены только операциями, которые допускаются соответствующими правилами вариантности.

Это также показывает, почему *классам* не позволено иметь вариантные параметры типа: конкретные реализации обычно требуют протекания данных в обоих направлениях.

Для другого примера необходим следующий интерфейс, который определен в .NET Framework:

```
public interface IComparer<in T>
{
  // Возвращает значение, отражающее относительный порядок a и b
  int Compare (T a, T b);
}
```

Поскольку интерфейс имеет контравариантный параметр типа T, мы можем применять IComparer<**object**> для сравнения двух *строк*:

```
var objectComparer = Comparer<object>.Default;
// objectComparer реализует IComparer<object>
IComparer<string> stringComparer = objectComparer;
int result = stringComparer.Compare ("Brett", "Jemaine");
```

Зеркально отражая ковариантность, компилятор сообщит об ошибке, если вы попытаетесь использовать контравариантный параметр типа в выходной позиции (например, в качестве возвращаемого значения или в читаемом свойстве).

## Сравнение обобщений C# и шаблонов C++

Обобщения C# в использовании похожи на шаблоны C++, но работают они совершенно по-другому. В обоих случаях должен осуществляться синтез между поставщиком и потребителем, при котором типы-заполнители заполняются потребителем. Однако в ситуации с обобщениями C# типы поставщика (т.е. открытые типы вроде `List<T>`) могут быть скомпилированы в библиотеку (такую как *mscorlib.dll*). Дело в том, что собственно синтез между поставщиком и потребителем, который создает закрытые типы, в действительности не происходит вплоть до времени выполнения. Для шаблонов C++ такой синтез производится на этапе компиляции. Это значит, что в C++ развертывать библиотеки шаблонов как сборки `.dll` не получится – они существуют только в виде исходного кода. Вдобавок также затрудняется динамическое инспектирование параметризированных типов, не говоря уже об их создании на лету.

Чтобы лучше понять, почему сказанное справедливо, взглянем на метод `Max` в C# еще раз:

```
static T Max <T> (T a, T b) where T : IComparable<T>
  => a.CompareTo (b) > 0 ? a : b;
```

Почему бы ни реализовать этот метод следующим образом:

```
static T Max <T> (T a, T b)
  => (a > b ? a : b);  // Ошибка на этапе компиляции
```

Причина в том, что метод `Max` должен быть скомпилирован один раз, но работать для всех возможных значений `T`. Компиляция не может пройти успешно ввиду отсутствия единого смысла операции > для всех значений `T` – на самом деле, операция > может быть доступна далеко не в каждом типе `T`. Для сравнения ниже показан код того же метода `Max`, написанный с применением шаблонов C++. Этот код будет компилироваться отдельно для каждого значения `T`, пользуясь семантикой > для конкретного типа `T` и приводя к ошибке на этапе компиляции, если отдельный тип `T` не поддерживает операцию >:

```
template <class T> T Max (T a, T b)
{
  return a > b ? a : b;
}
```

# 4

# Дополнительные средства C#

В настоящей главе мы раскроем более сложные темы, связанные с языком C#, которые построены на основе концепций, исследованных в главах 2 и 3. Первые четыре раздела необходимо читать последовательно, а остальные — в произвольном порядке.

## Делегаты

Делегат — это объект, которому известно, как вызывать метод.

*Тип делегата* определяет разновидность метода, которую могут вызывать *экземпляры делегата*. В частности, он определяет *возвращаемый тип* и *типы параметров* метода. Ниже показано определение типа делегата по имени `Transformer`:

```
delegate int Transformer (int x);
```

Делегат `Transformer` совместим с любым методом, который имеет возвращаемый тип `int` и принимает единственный параметр `int`, вроде следующего:

```
static int Square (int x) { return x * x; }
```

или более сжато:

```
static int Square (int x) => x * x;
```

Присваивание метода переменной делегата создает *экземпляр* делегата:

```
Transformer t = Square;
```

который можно вызывать тем же способом, что и метод:

```
int answer = t(3);  // answer получает значение 9
```

Вот завершенный пример:

```
delegate int Transformer (int x);

class Test
{
  static void Main()
  {
    Transformer t = Square;     // Создать экземпляр делегата
    int result = t(3);          // Вызвать экземпляр делегата
```

```
    Console.WriteLine (result);  // 9
  }
  static int Square (int x) => x * x;
}
```

Экземпляр делегата действует в вызывающем компоненте буквально как посредник: вызывающий компонент обращается к делегату, после чего делегат вызывает целевой метод. Такая косвенность отвязывает вызывающий компонент от целевого метода.

Оператор:

```
Transformer t = Square;
```

является сокращением следующего оператора:

```
Transformer t = new Transformer (Square);
```

Формально когда мы ссылаемся на `Square` без скобок или аргументов, то указываем *группу методов*. Если метод перегружен, тогда компилятор C# выберет корректную перегруженную версию на основе сигнатуры делегата, которому `Square` присваивается.

Выражение:

```
t(3)
```

представляет собой сокращение такого вызова:

```
t.Invoke(3)
```

Делегат похож на *обратный вызов* — общий термин, который охватывает конструкции вроде указателей на функции C.

## Написание подключаемых методов с помощью делегатов

Метод присваивается переменной делегата во время выполнения. Это удобно при написании подключаемых методов. В следующем примере присутствует служебный метод по имени `Transform`, который применяет трансформацию к каждому элементу в целочисленном массиве. Метод `Transform` имеет параметр делегата, предназначенный для указания подключаемой трансформации.

```
public delegate int Transformer (int x);

class Util
{
  public static void Transform (int[] values, Transformer t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }
}

class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
    Util.Transform (values, Square);  // Привязаться к методу Square
```

```
    foreach (int i in values)
      Console.Write (i + "  ");      // 1   4   9
  }

  static int Square (int x) => x * x;
}
```

Наш метод `Transform` является *функцией более высокого порядка*, потому что получает в качестве аргумента функцию. (Метод, который *возвращает* делегат, также будет функцией более высокого порядка.)

## Групповые делегаты

Все экземпляры делегатов обладают возможностью *группового вызова* (multicast). Это значит, что экземпляр делегата может ссылаться не только на одиночный целевой метод, но также и на список целевых методов. Экземпляры делегатов комбинируются с помощью операций + и +=. Например:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

Последняя строка функционально эквивалентна следующей строке:

```
d = d + SomeMethod2;
```

Обращение к `d` теперь приведет к вызову методов `SomeMethod1` и `SomeMethod2`. Делегаты вызываются в порядке, в котором они добавлялись.

Операции - и -= удаляют правый операнд делегата из левого операнда делегата. Например:

```
d -= SomeMethod1;
```

Обращение к `d` теперь приведет к вызову только метода `SomeMethod2`.

Использование операции + или += с переменной делегата, имеющей значение `null`, допустимо и эквивалентно присваиванию этой переменной нового значения:

```
SomeDelegate d = null;
d += SomeMethod1; //Эквивалентно (когда d равно null) оператору d=SomeMethod1;
```

Аналогичным образом применение операции -= к переменной делегата с единственным целевым методом эквивалентно присваиванию этой переменной значения `null`.

Делегаты являются *неизменяемыми*, так что в случае использования операции += или -= фактически создается *новый* экземпляр делегата и присваивается существующей переменной.

Если групповой делегат имеет возвращаемый тип, отличный от `void`, тогда вызывающий компонент получает возвращаемое значение из последнего вызванного метода. Предшествующие методы по-прежнему вызываются, но их возвращаемые значения отбрасываются. В большинстве сценариев применения групповые делегаты имеют возвращаемые типы `void`, так что описанная тонкая ситуация не возникает.

Все типы делегатов неявно порождены от класса `System.MulticastDelegate`, который унаследован от `System.Delegate`. Операции +, -, += и -=, выполняемые над делегатом, транслируются в вызовы статических методов `Combine` и `Remove` класса `System.Delegate`.

### Пример группового делегата

Предположим, что вы написали метод, выполнение которого занимает длительное время. Такой метод мог бы регулярно сообщать о ходе работ вызывающему компоненту, обращаясь к делегату. В следующем примере метод `HardWork` имеет параметр делегата `ProgressReporter`, который вызывается для отражения хода работ:

```
public delegate void ProgressReporter (int percentComplete);

public class Util
{
  public static void HardWork (ProgressReporter p)
  {
    for (int i = 0; i < 10; i++)
    {
      p (i * 10);                              // Вызвать делегат
      System.Threading.Thread.Sleep (100);  // Эмулировать длительную работу
    }
  }
}
```

Для мониторинга хода работ метод `Main` создает экземпляр группового делегата `p`, так что ход работ отслеживается двумя независимыми методами:

```
class Test
{
  static void Main()
  {
    ProgressReporter p = WriteProgressToConsole;
    p += WriteProgressToFile;
    Util.HardWork (p);
  }
  static void WriteProgressToConsole (int percentComplete)
    => Console.WriteLine (percentComplete);
  static void WriteProgressToFile (int percentComplete)
    => System.IO.File.WriteAllText ("progress.txt",
                                    percentComplete.ToString());
}
```

## Целевые методы экземпляра и целевые статические методы

Когда объекту делегата присваивается метод *экземпляра*, объект делегата должен поддерживать ссылку не только на метод, но также и на *экземпляр*, которому этот метод принадлежит. Экземпляр представлен свойством `Target` класса `System.Delegate` (которое будет равно `null`, если делегат ссылается на статический метод). Например:

```
public delegate void ProgressReporter (int percentComplete);

class Test
{
  static void Main()
  {
    X x = new X();
    ProgressReporter p = x.InstanceProgress;
    p(99);                                  // 99
    Console.WriteLine (p.Target == x);  // True
    Console.WriteLine (p.Method);       // void InstanceProgress(Int32)
  }
}
```

```
class X
{
  public void InstanceProgress (int percentComplete)
    => Console.WriteLine (percentComplete);
}
```

## Обобщенные типы делегатов

Тип делегата может содержать параметры обобщенного типа. Например:

```
public delegate T Transformer<T> (T arg);
```

Располагая таким определением, можно написать обобщенный служебный метод `Transform`, который работает с любым типом:

```
public class Util
{
  public static void Transform<T> (T[] values, Transformer<T> t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }
}

class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
    Util.Transform (values, Square);    // Привязаться к методу Square
    foreach (int i in values)
      Console.Write (i + "  ");         // 1   4   9
  }
  static int Square (int x) => x * x;
}
```

## Делегаты `Func` и `Action`

Благодаря обобщенным делегатам становится возможным написание небольшого набора типов делегатов, которые являются настолько универсальными, что могут работать с методами, имеющими любой возвращаемый тип и любое (разумное) количество аргументов. Такими делегатами являются `Func` и `Action`, определенные в пространстве имен `System` (модификаторы `in` и `out` указывают *вариантность*, которая вскоре будет раскрыта):

```
delegate TResult Func <out TResult>                ();
delegate TResult Func <in T, out TResult>          (T arg);
delegate TResult Func <in T1, in T2, out TResult>(T1 arg1, T2 arg2);
... и так далее вплоть до T16
delegate void Action                 ();
delegate void Action <in T>          (T arg);
delegate void Action <in T1, in T2> (T1 arg1, T2 arg2);
... и так далее вплоть до T16
```

Эти делегаты исключительно универсальны. Делегат `Transformer` в предыдущем примере может быть заменен делегатом `Func`, который принимает один аргумент типа `T` и возвращает значение того же самого типа:

```
public static void Transform<T> (T[] values, Func<T,T> transformer)
{
  for (int i = 0; i < values.Length; i++)
    values[i] = transformer (values[i]);
}
```

Делегаты `Func` и `Action` не покрывают лишь практические сценарии, связанные с параметрами `ref/out` и параметрами указателей.

До выхода версии .NET Framework 2.0 делегаты `Func` и `Action` не существовали (поскольку не было обобщений). Именно по указанной исторической причине в большей части .NET Framework используются специальные типы делегатов, а не `Func` и `Action`.

## Сравнение делегатов и интерфейсов

Задачу, которую можно решить с помощью делегата, вполне реально решить также посредством интерфейса. Мы можем переписать исходный пример, применяя вместо делегата интерфейс `ITransformer`:

```
public interface ITransformer
{
  int Transform (int x);
}

public class Util
{
 public static void TransformAll (int[] values, ITransformer t)
 {
   for (int i = 0; i < values.Length; i++)
     values[i] = t.Transform (values[i]);
 }
}

class Squarer : ITransformer
{
  public int Transform (int x) => x * x;
}
...
static void Main()
{
  int[] values = { 1, 2, 3 };
  Util.TransformAll (values, new Squarer());
  foreach (int i in values)
    Console.WriteLine (i);
}
```

Решение на основе делегатов может оказаться более удачным, чем решение на основе интерфейсов, если соблюдено одно или более следующих условий:

- в интерфейсе определен только один метод;
- требуется возможность группового вызова;
- подписчик нуждается в многократной реализации интерфейса.

В примере с `ITransformer` групповой вызов не нужен. Однако в интерфейсе определен только один метод. Более того, подписчику может потребоваться многократ-

ная реализация интерфейса ITransformer для поддержки разнообразных трансформаций наподобие возведения в квадрат или куб. В случае интерфейсов нам придется писать отдельный тип для каждой трансформации, т.к. класс может реализовывать ITransformer только один раз. В результате получается довольно громоздкий код:

```
class Squarer : ITransformer
{
  public int Transform (int x) => x * x;
}
class Cuber : ITransformer
{
  public int Transform (int x) => x * x * x;
}
...
static void Main()
{
  int[] values = { 1, 2, 3 };
  Util.TransformAll (values, new Cuber());
  foreach (int i in values)
    Console.WriteLine (i);
}
```

# Совместимость делегатов

## Совместимость типов

Все типы делегатов несовместимы друг с другом, даже если они имеют одинаковые сигнатуры:

```
delegate void D1();
delegate void D2();
...
D1 d1 = Method1;
D2 d2 = d1;  // Ошибка на этапе компиляции
```

Однако следующий код разрешен:

```
D2 d2 = new D2 (d1);
```

Экземпляры делегатов считаются равными, если они имеют одинаковые целевые методы:

```
delegate void D();
...
D d1 = Method1;
D d2 = Method1;
Console.WriteLine (d1 == d2);  // True
```

Групповые делегаты считаются равными, если они ссылаются на те же самые методы *в одинаковом порядке.*

## Совместимость параметров

При вызове метода можно предоставлять аргументы, которые относятся к более специфичным типам, нежели те, что определены для параметров данного метода. Это обычное полиморфное поведение. По той же причине делегат может иметь более специфичные типы параметров, чем его целевой метод. Это называется *контравариантностью.*

Вот пример:

```
delegate void StringAction (string s);

class Test
{
  static void Main()
  {
    StringAction sa = new StringAction (ActOnObject);
    sa ("hello");
  }

  static void ActOnObject (object o) => Console.WriteLine (o);  // hello
}
```

(Как и с вариантностью параметров типа, делегаты являются вариантными только для *ссылочных преобразований.*)

Делегат просто вызывает метод от имени кого-то другого. В таком случае `StringAction` вызывается с аргументом типа `string`. Когда аргумент затем передается целевому методу, он неявно приводится вверх к `object`.

Стандартный шаблон событий спроектирован для того, чтобы помочь задействовать контравариантность через использование общего базового класса `EventArgs`. Например, можно иметь единственный метод, который вызывается двумя разными делегатами с передачей одному объекта `MouseEventArgs`, а другому – `KeyEventArgs`.

### Совместимость возвращаемых типов

В результате вызова метода можно получить обратно тип, который является более специфическим, чем запрошенный. Так выглядит обычное полиморфное поведение. По той же самой причине целевой метод делегата может возвращать более специфический тип, чем описанный самим делегатом. Это называется *ковариантностью.* Например:

```
delegate object ObjectRetriever();

class Test
{
  static void Main()
  {
    ObjectRetriever o = new ObjectRetriever (RetrieveString);
    object result = o();
    Console.WriteLine (result);  // hello
  }
  static string RetrieveString() => "hello";
}
```

Делегат `ObjectRetriever` ожидает получить обратно `object`, но может быть получен также и *подкласс* `object`, потому что возвращаемые типы делегатов являются *ковариантными.*

### Вариантность параметров типа обобщенного делегата

В главе 3 было показано, что обобщенные интерфейсы поддерживают ковариантные и контравариантные параметры типа. Та же самая возможность существует и для делегатов (начиная с версии C# 4.0).

При определении обобщенного типа делегата рекомендуется поступать следующим образом:

- помечать параметр типа, применяемый только для возвращаемого значения, как ковариантный (`out`);
- помечать любой параметр типа, используемый только для параметров, как контравариантный (`in`).

В результате преобразования смогут работать естественным образом, соблюдая отношения наследования между типами.

Показанный ниже делегат (определенный в пространстве имен `System`) имеет ковариантный параметр `TResult`:

```
delegate void Action<in T> (T arg);
```

позволяя записывать так:

```
Func<string> x = ...;
Func<object> y = x;
```

Следующий делегат (определенный в пространстве имен `System`) имеет контравариантный параметр `T`:

```
delegate void Action<in T> (T arg);
```

делая возможным такой код:

```
Action<object> x = ...;
Action<string> y = x;
```

# События

Во время применения делегатов обычно возникают две независимые роли: *ретранслятор* и *подписчик*.

*Ретранслятор* – это тип, который содержит поле делегата. Ретранслятор решает, когда делать передачу, вызывая делегат.

*Подписчики* – это целевые методы-получатели. Подписчик решает, когда начинать и останавливать прослушивание, используя операции `+=` и `-=` на делегате ретранслятора. Подписчик ничего не знает о других подписчиках и не вмешивается в их работу.

События являются языковым средством, которое формализует описанный шаблон. Конструкция `event` открывает только подмножество возможностей делегата, требуемое для модели "ретранслятор/подписчик". Основное назначение событий заключается в *предотвращении влияния подписчиков друг на друга*.

Простейший способ объявления события предусматривает помещение ключевого слова `event` перед членом делегата:

```
// Определение делегата
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Broadcaster
{
  // Объявление события
  public event PriceChangedHandler PriceChanged;
}
```

Код внутри типа `Broadcaster` имеет полный доступ к члену `PriceChanged` и может трактовать его как делегат. Код за пределами `Broadcaster` может только выполнять операции `+=` и `-=` над событием `PriceChanged`.

## Внутренняя работа событий

При объявлении показанного ниже события происходят три действия:

```
public class Broadcaster
{
  public event PriceChangedHandler PriceChanged;
}
```

Во-первых, компилятор транслирует объявление события в примерно такой код:

```
PriceChangedHandler priceChanged;  // закрытый делегат
public event PriceChangedHandler PriceChanged
{
  add    { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

Ключевыми словами `add` и `remove` обозначаются явные *средства доступа к событию*, которые работают аналогично средствам доступа к свойству. Позже мы покажем, как их реализовать.

Во-вторых, компилятор ищет *внутри* класса `Broadcaster` ссылки на `PriceChanged`, в которых выполняются операции, отличные от `+=` или `-=`, и переадресует их на лежащее в основе поле делегата `priceChanged`.

В-третьих, компилятор транслирует операции `+=` и `-=`, примененные к событию, в обращения к средствам доступа `add` и `remove` события. Интересно, что это делает поведение операций `+=` и `-=` уникальным в случае применения к событиям: в отличие от других сценариев они не являются просто сокращением для операций `+` и `-`, за которыми следует операция присваивания.

Рассмотрим следующий пример. Класс `Stock` запускает свое событие `PriceChanged` каждый раз, когда изменяется свойство `Price` данного класса:

```
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event PriceChangedHandler PriceChanged;

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;        // Выйти, если ничего не изменялось
      decimal oldPrice = price;
      price = value;
      if (PriceChanged != null)          // Если список вызова не пуст,
        PriceChanged (oldPrice, price); // тогда запустить событие
    }
  }
}
```

Если в приведенном примере убрать ключевое слово `event`, чтобы `PriceChanged` превратилось в обычное поле делегата, то результаты окажутся теми же самыми. Но класс `Stock` станет менее надежным в том, что подписчики смогут предпринимать следующие действия, влияя друг на друга:

- заменять других подписчиков, переустанавливая `PriceChanged` (вместо использования операции +=);
- очищать всех подписчиков (устанавливая `PriceChanged` в `null`);
- выполнять групповую рассылку другим подписчикам путем вызова делегата.

События WinRT имеют слегка отличающуюся семантику, которая заключается в том, что присоединение к событию возвращает маркер, требующийся для отсоединения от события в будущем. Компилятор прозрачно устраняет указанную брешь (за счет поддержки внутреннего словаря маркеров), поэтому события WinRT можно использовать так, как если бы они были обычными событиями CLR.

## Стандартный шаблон событий

В .NET Framework определен стандартный шаблон для написания событий. Его целью является обеспечение согласованности между .NET Framework и пользовательским кодом. В основе стандартного шаблона событий находится `System.EventArgs` — предопределенный класс .NET Framework, имеющий единственное статическое свойство `Empty`. Базовый класс `EventArgs` предназначен для передачи информации событию. В рассматриваемом примере `Stock` мы создаем подкласс `EventArgs` для передачи старого и нового значений цены, когда инициируется событие `PriceChanged`:

```
public class PriceChangedEventArgs : System.EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice;
    NewPrice = newPrice;
  }
}
```

Для обеспечения многократного использования подкласс `EventArgs` именован в соответствии с содержащейся в нем информацией (а не с событием, для которого он будет применяться). Обычно он открывает доступ к данным как к свойствам или полям, предназначенным только для чтения.

Имея подкласс `EventArgs`, далее потребуется выбрать или определить делегат для события согласно следующим трем правилам.

- Он должен иметь возвращаемый тип `void`.
- Он должен принимать два аргумента: первый — `object` и второй — подкласс `EventArgs`. Первый аргумент указывает ретранслятор события, а второй аргумент содержит дополнительную информацию для передачи событию.
- Его имя должно заканчиваться на *`EventHandler`*.

В .NET Framework определен обобщенный делегат по имени `System.EventHandler<>`, который удовлетворяет описанным правилам:

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e) where TEventArgs : EventArgs;
```

До появления в языке обобщений (до выхода версии C# 2.0) взамен необходимо было записывать специальный делегат следующего вида:

```
public delegate void PriceChangedHandler
  (object sender, PriceChangedEventArgs e);
```

По историческим причинам большинство событий в .NET Framework используют делегаты, определенные подобным образом.

Следующий шаг – определение события выбранного типа делегата. В приведенном ниже коде применяется обобщенный делегат `EventHandler`:

```
public class Stock
{
  ...
  public event EventHandler<PriceChangedEventArgs> PriceChanged;
}
```

Наконец, шаблон требует написания защищенного виртуального метода, который запускает событие. Имя такого метода должно совпадать с именем события, предваренным словом *On*, и он должен принимать единственный аргумент `EventArgs`:

```
public class Stock
{
  ...
  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged (PriceChangedEventArgs e)
  {
    if (PriceChanged != null) PriceChanged (this, e);
  }
}
```

В многопоточных сценариях (глава 14) перед проверкой и вызовом делегат необходимо присваивать временной переменной во избежание ошибки, связанной с безопасностью потоков:

```
var temp = PriceChanged;
if (temp != null) temp (this, e);
```

Начиная с версии C# 6, ту же самую функциональность можно получить без переменной `temp` с помощью `null`-условной операции:

```
PriceChanged?.Invoke (this, e);
```

Будучи безопасным к потокам и лаконичным, теперь это наилучший общепринятый способ вызова событий.

В результате мы имеем центральную точку, из которой подклассы могут вызывать или переопределять событие (предполагая, что класс не запечатан).

Ниже приведен завершенный код примера:

```
using System;

public class PriceChangedEventArgs : EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice; NewPrice = newPrice;
  }
}

public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) {this.symbol = symbol;}

  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged (PriceChangedEventArgs e)
  {
    PriceChanged?.Invoke (this, e);
  }

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      decimal oldPrice = price;
      price = value;
      OnPriceChanged (new PriceChangedEventArgs (oldPrice, price));
    }
  }
}

class Test
{
  static void Main()
  {
    Stock stock = new Stock ("THPW");
    stock.Price = 27.10M;
    // Зарегистрировать с событием PriceChanged
    stock.PriceChanged += stock_PriceChanged;
    stock.Price = 31.59M;
  }

  static void stock_PriceChanged (object sender, PriceChangedEventArgs e)
  {
    if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
      // Сигнал об увеличении цены на 10%
      Console.WriteLine ("Alert, 10% stock price increase!");
  }
}
```

Когда событие не несет в себе дополнительную информацию, можно использовать предопределенный необобщенный делегат `EventHandler`. Мы перепишем код класса `Stock` так, чтобы событие `PriceChanged` запускалось после изменения цены, причем какая-либо информация о событии не требуется — необходимо сообщить лишь о самом факте его возникновения. Мы также будем применять свойство `EventArgs.Empty`, чтобы избежать ненужного создания экземпляра `EventArgs`.

```
public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event EventHandler PriceChanged;

  protected virtual void OnPriceChanged (EventArgs e)
  {
    PriceChanged?.Invoke (this, e);
  }

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      price = value;
      OnPriceChanged (EventArgs.Empty);
    }
  }
}
```

## Средства доступа к событию

*Средства доступа* к событию представляют собой реализации его операций += и -=. По умолчанию средства доступа реализуются неявно компилятором. Взгляните на следующее объявление события:

```
public event EventHandler PriceChanged;
```

Компилятор преобразует его в перечисленные ниже компоненты:

- закрытое поле делегата;
- пара открытых функций доступа к событию (`add_PriceChanged` и `remove_PriceChanged`), реализации которых переадресуют операции += и -= закрытому полю делегата.

Контроль над процессом преобразования можно взять на себя, определив *явные* средства доступа. Вот как выглядит ручная реализация события `PriceChanged` из предыдущего примера:

```
private EventHandler priceChanged;  // Объявить закрытый делегат

public event EventHandler PriceChanged
{
  add    { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

Приведенный пример функционально идентичен стандартной реализации средств доступа C# (за исключением того, что C# также обеспечивает безопасность в отношении потоков во время обновления делегата через свободный от блокировок алгоритм сравнения и обмена – см. http://albahari.com/threading). Определяя средства доступа к событию самостоятельно, мы указываем C# на то, что генерировать стандартное поле и логику средств доступа не требуется.

С помощью явных средств доступа к событию можно реализовать более сложные стратегии хранения и обращения к лежащему в основе делегату. Ниже описаны три сценария, в которых это полезно.

- Когда средства доступа к событию просто поручают другому классу групповую передачу события.
- Когда класс открывает доступ к большому количеству событий, для которых бо́льшую часть времени существует очень мало подписчиков, как в случае элемента управления Windows. В таких ситуациях лучше хранить экземпляры делегатов подписчиков в словаре, т.к. со словарем связаны меньшие накладные расходы по хранению, чем с десятками нулевых ссылок на поля делегатов.
- Когда явно реализуется интерфейс, в котором объявлено событие.

Рассмотрим пример, иллюстрирующий последний сценарий:

```
public interface IFoo { event EventHandler Ev; }

class Foo : IFoo
{
  private EventHandler ev;

  event EventHandler IFoo.Ev
  {
    add    { ev += value; }
    remove { ev -= value; }
  }
}
```

Части add и remove события транслируются в методы add_*XXX* и remove_*XXX*.

## Модификаторы событий

Подобно методам события могут быть виртуальными, переопределенными, абстрактными или запечатанными. События также могут быть статическими:

```
public class Foo
{
  public static event EventHandler<EventArgs> StaticEvent;
  public virtual event EventHandler<EventArgs> VirtualEvent;
}
```

# Лямбда-выражения

Лямбда-выражение – это неименованный метод, записанный вместо экземпляра делегата. Компилятор немедленно преобразует лямбда-выражение в одну из следующих двух конструкций.

- Экземпляр делегата.
- *Дерево выражения*, которое имеет тип `Expression<TDelegate>` и представляет код внутри лямбда-выражения в виде поддерживающей обход объектной модели. Дерево выражения позволяет лямбда-выражению интерпретироваться позже во время выполнения (см. раздел "Построение выражений запросов" в главе 8).

Имея показанный ниже тип делегата:

```
delegate int Transformer (int i);
```

вот как можно присвоить и обратиться к лямбда-выражению `x => x * x`:

```
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3));  // 9
```

Внутренне компилятор преобразует лямбда-выражение данного типа в закрытый метод, телом которого будет код выражения.

Лямбда-выражение имеет следующую форму:

```
(параметры) => выражение-или-блок-операторов
```

Для удобства круглые скобки можно опускать, но только в ситуации, когда есть в точности один параметр выводимого типа.

В нашем примере присутствует единственный параметр `x`, а выражением является `x * x`:

```
x => x * x;
```

Каждый параметр лямбда-выражения соответствует параметру делегата, а тип выражения (которым может быть `void`) — возвращаемому типу этого делегата.

В данном примере `x` соответствует параметру `i`, а выражение `x * x` — возвращаемому типу `int` и потому оно совместимо с делегатом `Transformer`:

```
delegate int Transformer (int i);
```

Код лямбда-выражения может быть *блоком операторов*, а не выражением. Мы можем переписать пример следующим образом:

```
x => { return x * x; };
```

Лямбда-выражения чаще всего применяются с делегатами `Func` и `Action`, так что приведенное ранее выражение вы будете регулярно встречать в такой форме:

```
Func<int,int> sqr = x => x * x;
```

Ниже показан пример выражения, которое принимает два параметра:

```
Func<string,string,int> totalLength = (s1, s2) => s1.Length + s2.Length;
int total = totalLength ("hello", "world");  // total равно 10;
```

Лямбда-выражения появились в версии C# 3.0.

## Явное указание типов лямбда-параметров

Компилятор обычно способен *выводить* типы лямбда-параметров из контекста. Когда это не так, вы должны явно указать тип для каждого параметра. Взгляните на следующие два метода:

```
void Foo<T> (T x)        {}
void Bar<T> (Action<T> a) {}
```

Приведенный далее код не скомпилируется, потому что компилятор не сможет вывести тип x:

```
Bar (x => Foo (x));      // К какому типу относится x?
```

Исправить ситуацию можно явным указанием типа x следующим образом:

```
Bar ((int x) => Foo (x));
```

Рассматриваемый пример довольно прост и может быть исправлен еще двумя способами:

```
Bar<int> (x => Foo (x)); // Указать параметр типа для Bar
Bar<int> (Foo);          // Как и выше, но использовать группу методов
```

## Захватывание внешних переменных

Лямбда-выражение может ссылаться на локальные переменные и параметры метода, в котором оно определено (*внешние переменные*). Например:

```
static void Main()
{
  int factor = 2;
  Func<int, int> multiplier = n => n * factor;
  Console.WriteLine (multiplier (3));  // 6
}
```

Внешние переменные, на которые ссылается лямбда-выражение, называются *захваченными переменными*. Лямбда-выражение, захватывающее переменные, называется *замыканием*.

Переменные могут захватываться также анонимными методами и локальными методами. Во всех случаях по отношению к захваченным переменным действуют те же самые правила.

Захваченные переменные оцениваются, когда делегат фактически *вызывается*, а не когда переменные были *захвачены*:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine (multiplier (3));    // 30
```

Лямбда-выражения сами могут обновлять захваченные переменные:

```
int seed = 0;
Func<int> natural = () => seed++;
Console.WriteLine (natural());         // 0
Console.WriteLine (natural());         // 1
Console.WriteLine (seed);              // 2
```

Захваченные переменные имеют свое время жизни, расширенное до времени жизни делегата. В следующем примере локальная переменная `seed` обычно покидала бы область видимости после того, как выполнение `Natural` завершено. Но поскольку переменная `seed` была *захвачена*, время жизни этой переменной расширяется до времени жизни захватившего ее делегата, т.е. `natural`:

```
static Func<int> Natural()
{
  int seed = 0;
  return () => seed++;  // Возвращает замыкание
}
```

```
static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());  // 0
  Console.WriteLine (natural());  // 1
}
```

Локальная переменная, *созданная* внутри лямбда-выражения, будет уникальной для каждого вызова экземпляра делегата. Если мы переделаем предыдущий пример, чтобы создавать seed внутри лямбда-выражения, то получим разные (в данном случае нежелательные) результаты:

```
static Func<int> Natural()
{
  return() => { int seed = 0; return seed++; };
}

static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());  // 0
  Console.WriteLine (natural());  // 0
}
```

Внутренне захватывание реализуется "заимствованием" захваченных переменных и помещением их в поля закрытого класса. При вызове метода создается экземпляр этого класса и привязывается на время жизни к экземпляру делегата.

### Захватывание итерационных переменных

Когда захватывается итерационная переменная в цикле `for`, она трактуется компилятором C# так, как если бы она была объявлена *вне* цикла. Таким образом, на каждой итерации захватывается *та же самая* переменная. Приведенный ниже код выводит 333, а не 012:

```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
  actions [i] = () => Console.Write (i);

foreach (Action a in actions) a();  // 333
```

Каждое замыкание (выделенное полужирным) захватывает одну и ту же переменную `i`. (Это действительно имеет смысл, когда считать, что `i` является переменной, значение которой сохраняется между итерациями цикла; при желании `i` можно даже явно изменять внутри тела цикла.) Последствие заключается в том, что при более позднем вызове каждый делегат видит значение `i` на момент *вызова*, т.е. 3. Чтобы лучше проиллюстрировать сказанное, развернем цикл `for`:

```
Action[] actions = new Action[3];
int i = 0;
actions[0] = () => Console.Write (i);
i = 1;
actions[1] = () => Console.Write (i);
i = 2;
actions[2] = () => Console.Write (i);
i = 3;
foreach (Action a in actions) a();  // 333
```

Если требуется вывести на экран 012, то решение состоит в том, чтобы присвоить итерационную переменную какой-то локальной переменной с областью видимости *внутри* цикла:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
{
  int loopScopedi = i;
  actions [i] = () => Console.Write (loopScopedi);
}
foreach (Action a in actions) a();  // 012
```

Из-за того, что во время любой итерации переменная loopScopedi создается заново, каждое замыкание захватывает *отличающуюся* переменную.

До появления версии C# 5.0 циклы foreach работали аналогично:

```
Action[] actions = new Action[3];
int i = 0;
foreach (char c in "abc")
  actions [i++] = () => Console.Write (c);
foreach (Action a in actions) a();  // Выводит ccc в версии C# 4.0
```

В результате возникала значительная путаница: в отличие от цикла for итерационная переменная в цикле foreach неизменяема, и можно было бы ожидать, что она трактуется как локальная по отношению к телу цикла. Хорошая новость в том, что в версии C# 5.0 проблема была исправлена, а потому показанный выше пример теперь выводит abc.

Формально это является критическим изменением, поскольку перекомпиляция программы, написанной на C# 4.0, в C# 5.0 может привести к получению других результатов. В целом команда разработчиков C# старается избегать критических изменений; однако в данном случае "критичность", несомненно, связана с исправлением необнаруженной ошибки в программе C# 4.0, а не с преднамеренной опорой на старое поведение.

## Сравнение лямбда-выражений и локальных методов

Функциональность локальных методов C# 7 (см. раздел "Локальные методы" в главе 1) частично совпадает с функциональностью лямбда-выражений. Локальные методы обладают следующими тремя преимуществами:

- они могут быть рекурсивными (вызывать сами себя) без неуклюжих трюков;
- они лишены беспорядка, связанного с указанием типа делегата;
- они требуют чуть меньших накладных расходов.

Локальные методы более эффективны, потому что избегают косвенности делегата (за которую приходится платить дополнительными циклами центрального процессора и выделением памяти). Они также могут получать доступ к локальным переменным содержащего метода, не заставляя компилятор "заимствовать" захваченные переменные и помещать их в скрытый класс.

Тем не менее, во многих случаях делегат *необходим*, чаще всего при вызове функции высокого порядка, т.е. метода с параметром типа делегата:

```
public void Foo (Func<int,bool> predicate) { ... }
```

(Дополнительные сведения ищите в главе 8.) В ситуациях подобного рода, так или иначе, необходим делегат, и они представляют собой в точности те случаи, когда лямбда-выражения обычно короче и яснее.

# Анонимные методы

*Анонимные методы* – это функциональная возможность C# 2.0, которая по большей части относится к лямбда-выражениям. Анонимный метод похож на лямбда-выражение, но в нем отсутствуют:

- неявно типизированные параметры;
- синтаксис выражений (анонимный метод должен всегда быть блоком операторов);
- возможность компиляции в дерево выражения путем присваивания объекту типа `Expression<T>`.

Чтобы написать анонимный метод, понадобится указать ключевое слово `delegate`, далее (необязательное) объявление параметра и затем тело метода. Например, имея следующий делегат:

```
delegate int Transformer (int i);
```

мы можем написать и вызвать анонимный метод, как показано ниже:

```
Transformer sqr = delegate (int x) {return x * x;};
Console.WriteLine (sqr(3));  // 9
```

Первая строка семантически эквивалентна следующему лямбда-выражению:

```
Transformer sqr =      (int x) => {return x * x;};
```

Или просто:

```
Transformer sqr =           x  => x * x;
```

Анонимные методы захватывают внешние переменные тем же самым способом, что и лямбда-выражения.

Уникальной особенностью анонимных методов является возможность полностью опускать объявление параметра – даже если делегат его ожидает. Это может быть удобно при объявлении событий со стандартным пустым обработчиком:

```
public event EventHandler Clicked = delegate { };
```

В итоге устраняется необходимость проверки на равенство `null` перед запуском события. Приведенный далее код также будет допустимым:

```
// Обратите внимание, что параметры не указаны:
Clicked += delegate { Console.WriteLine ("clicked"); };
```

# Операторы `try` и исключения

Оператор `try` указывает блок кода, предназначенный для обработки ошибок или очистки. За *блоком* `try` должен следовать *блок* `catch`, *блок* `finally` или оба. Блок `catch` выполняется, когда случается ошибка в блоке `try`. Блок `finally` выполняется после выполнения блока `try` (либо блока `catch`, если он присутствует), обеспечивая очистку независимо от того, возникала ошибка или нет.

Блок `catch` имеет доступ к объекту `Exception`, который содержит информацию об ошибке. Блок `catch` применяется либо для компенсации последствий ошибки, либо для *повторной генерации* исключения. Исключение генерируется повторно, если требуется просто зарегистрировать факт возникновения проблемы в журнале или если необходимо сгенерировать исключение нового более высокоуровневого типа.

Блок `finally` добавляет к программе детерминизма: среда CLR старается выполнять его всегда. Он полезен для проведения задач очистки вроде закрытия сетевых подключений. Оператор `try` выглядит следующим образом:

```
try
{
  ... // Во время выполнения этого блока может возникнуть исключение
}
catch (ExceptionA ex)
{
  ... // Обработать исключение типа ExceptionA
}
catch (ExceptionB ex)
{
  ... // Обработать исключение типа ExceptionB
}
finally
{
  ... // Код очистки
}
```

Взгляните на показанный далее код:

```
class Test
{
  static int Calc (int x) => 10 / x;
  static void Main()
  {
    int y = Calc (0);
    Console.WriteLine (y);
  }
}
```

Поскольку x имеет нулевое значение, исполняющая среда генерирует исключение `DivideByZeroException` и программа завершается. Чтобы предотвратить такое поведение, мы перехватываем исключение:

```
class Test
{
  static int Calc (int x) => 10 / x;
  static void Main()
  {
    try
    {
      int y = Calc (0);
      Console.WriteLine (y);
    }
    catch (DivideByZeroException ex)
    {
      Console.WriteLine ("x cannot be zero"); //значение x не может быть равно 0
    }
    Console.WriteLine ("program completed"); // программа завершена
  }
}
```

```
ВЫВОД:
x cannot be zero
program completed
```

Приведенный простой пример предназначен только для демонстрации обработки исключений. На практике вместо реализации такого сценария лучше перед вызовом `Calc` явно проверять делитель на равенство нулю.

Проверка с целью предотвращения ошибок предпочтительнее реализации блоков `try/catch`, т.к. обработка исключений является относительно затратной в плане ресурсов, требуя немало процессорного времени.

Когда возникает исключение, среда CLR выполняет следующую проверку.

*Находится ли поток выполнения в текущий момент внутри оператора* `try`*, который может перехватить исключение?*

- Если да, то поток выполнения переходит к совместимому блоку `catch`. Если этот блок `catch` завершился успешно, тогда поток выполнения перемещается на оператор, следующий после `try` (сначала выполнив блок `finally` при его наличии).
- Если нет, то поток выполнения возвращается обратно в вызывающий компонент и проверка повторяется (после выполнения любых блоков `finally`, внутри которых находится оператор).

Если ни одна из функций в стеке вызовов не взяла на себя ответственность за исключение, тогда пользователю отображается диалоговое окно с сообщением об ошибке и программа завершается.

## Конструкция `catch`

Конструкция `catch` указывает тип исключения, подлежащего перехвату. Типом может быть либо класс `System.Exception`, либо какой-то подкласс `System.Exception`.

Указание типа `System.Exception` приводит к перехвату всех возможных ошибок. Это удобно в следующих ситуациях:

- программа потенциально может восстановиться независимо от конкретного типа исключения;
- планируется повторная генерация исключения (возможно, после его регистрации в журнале);
- обработчик ошибок является последним средством перед тем, как программа прекратит работу.

Однако более обычной является ситуация, когда перехватываются *исключения специфических типов*, чтобы не иметь дела с исключениями, на которые обработчик не был рассчитан (например, `OutOfMemoryException`).

Перехватывать исключения нескольких типов можно с помощью множества конструкций `catch` (и снова данный пример проще реализовать посредством явной проверки аргументов, а не за счет обработки исключений):

```
class Test
{
  static void Main (string[] args)
  {
```

```
    try
    {
      byte b = byte.Parse (args[0]);
      Console.WriteLine (b);
    }
    catch (IndexOutOfRangeException ex)
    {
      // Должен быть предоставлен хотя бы один аргумент
      Console.WriteLine ("Please provide at least one argument");
    }
    catch (FormatException ex)
    {
      // Аргумент должен быть числовым
      Console.WriteLine ("That's not a number!");
    }
    catch (OverflowException ex)
    {
      // Возникло переполнение
      Console.WriteLine ("You've given me more than a byte!");
    }
  }
}
```

Для заданного исключения выполняется только одна конструкция `catch`. Если вы хотите предусмотреть сетку безопасности для перехвата общих исключений (вроде `System.Exception`), то должны размещать более специфические обработчики *первыми*.

Исключение может быть перехвачено без указания переменной, если доступ к свойствам исключения не нужен:

```
catch (OverflowException)  // переменная не указана
{
  ...
}
```

Кроме того, можно опускать и переменную, и тип (тогда будут перехватываться все исключения):

```
catch { ... }
```

### Фильтры исключений (C# 6)

Начиная с версии C# 6.0, в конструкции `catch` можно указывать *фильтр исключений* с помощью конструкции `when`:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
  ...
}
```

Если в приведенном примере генерируется исключение `WebException`, тогда вычисляется булевское выражение, находящееся после ключевого слова `when`. Если результатом оказывается `false`, то данный блок `catch` игнорируется и просматриваются любые последующие конструкции `catch`. Благодаря фильтрам исключений может появиться смысл в повторном перехвате исключения того же самого типа:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{ ... }
catch (WebException ex) when (ex.Status == WebExceptionStatus.SendFailure)
{ ... }
```

Булевское выражение в конструкции when может иметь побочные эффекты, например, вызывать метод, который фиксирует в журнале сведения об исключении для целей диагностики.

## Блок finally

Блок finally выполняется всегда – независимо от того, возникало ли исключение, и полностью ли был выполнен блок try. Блоки finally обычно используются для размещения кода очистки.

Блок finally выполняется в любом из следующих случаев:

- после завершения блока catch;
- после того, как поток управления покидает блок try из-за наличия оператора перехода (например, return или goto);
- после завершения блока try.

Единственное, что может воспрепятствовать выполнению блока finally – бесконечный цикл или неожиданное завершение процесса.

Блок finally содействует повышению детерминизма программы. В показанном ниже примере открываемый файл *всегда* закрывается независимо от перечисленных далее обстоятельств:

- блок try завершается нормально;
- происходит преждевременный возврат из-за того, что файл пуст (EndOfStream);
- во время чтения файла возникает исключение IOException.

```
static void ReadFile()
{
  StreamReader reader = null;  // Из пространства имен System.IO
  try
  {
    reader = File.OpenText ("file.txt");
    if (reader.EndOfStream) return;
    Console.WriteLine (reader.ReadToEnd());
  }
  finally
  {
    if (reader != null) reader.Dispose();
  }
}
```

Здесь мы закрываем файл с помощью вызова Dispose на StreamReader. Вызов Dispose на объекте внутри блока finally представляет собой стандартное соглашение, повсеместно соблюдаемое в .NET Framework, и оно явно поддерживается в языке C# через оператор using.

### Оператор using

Многие классы инкапсулируют неуправляемые ресурсы, такие как файловые и графические дескрипторы или подключения к базам данных. Классы подобного рода реализуют интерфейс System.IDisposable, в котором определен единственный метод без параметров по имени Dispose, предназначенный для очистки этих ресурсов. Оператор using предлагает элегантный синтаксис для вызова Dispose на объекте реализации IDisposable внутри блока finally.

Оператор:

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
  ...
}
```

в точности эквивалентен следующему коду:

```
{
  StreamReader reader = File.OpenText ("file.txt");
  try
  {
    ...
  }
  finally
  {
    if (reader != null)
      ((IDisposable)reader).Dispose();
  }
}
```

Шаблон освобождаемых объектов более подробно рассматривается в главе 12.

## Генерация исключений

Исключения могут генерироваться либо исполняющей средой, либо пользовательским кодом. В приведенном далее примере метод `Display` генерирует исключение `System.ArgumentNullException`:

```
class Test
{
  static void Display (string name)
  {
    if (name == null)
      throw new ArgumentNullException (nameof (name));

    Console.WriteLine (name);
  }

  static void Main()
  {
    try { Display (null); }
    catch (ArgumentNullException ex)
    {
      Console.WriteLine ("Caught the exception");  // Исключение перехвачено
    }
  }
}
```

### Выражения `throw`

До выхода версии C# 7 конструкция `throw` всегда была оператором. Теперь она может также появляться как выражение в функциях, сжатых до выражения:

```
public string Foo() => throw new NotImplementedException();
```

Выражение `throw` может также находиться внутри тернарной условной операции:

```
string ProperCase (string value) =>
  value == null ? throw new ArgumentException ("value") :
  value == "" ? "" :
  char.ToUpper (value[0]) + value.Substring (1);
```

### Повторная генерация исключения

Исключение можно перехватывать и генерировать повторно:

```
try { ... }
catch (Exception ex)
{
  // Записать в журнал информацию об ошибке
  ...
  throw;  // Повторно сгенерировать то же самое исключение
}
```

Если `throw` заменить `throw ex`, то пример сохранит работоспособность, но свойство `StackTrace` исключения больше не будет отражать исходную ошибку.

Повторная генерация в подобной манере дает возможность зарегистрировать в журнале информацию об ошибке, не *подавляя* ее. Она также позволяет отказаться от обработки исключения, если обстоятельства сложились не так, как ожидалось.

```
using System.Net;  // (См. главу 16.)
...
string s = null;
using (WebClient wc = new WebClient())
  try { s = wc.DownloadString ("http://www.albahari.com/nutshell/"); }
  catch (WebException ex)
  {
    if (ex.Status == WebExceptionStatus.Timeout)
      Console.WriteLine ("Timeout");
    else
      throw;  // Нет возможности обработать другие виды WebException,
              // поэтому сгенерировать исключение повторно
  }
```

Начиная с версии C# 6.0, код можно записать более лаконично с применением фильтра исключений:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
  Console.WriteLine ("Timeout");
}
```

Еще один распространенный сценарий предусматривает повторную генерацию исключения более специфического или содержательного типа. Например:

```
try
{
  ... // Получить значение DateTime из данных XML-элемента
}
catch (FormatException ex)
{
  throw new XmlException ("Invalid DateTime", ex);    // Недопустимый
                                                      // формат DateTime
}
```

Обратите внимание, что при создании экземпляра `XmlException` мы передаем конструктору во втором аргументе исходное исключение `ex`. Этот аргумент заполняет свойство `InnerException` нового экземпляра исключения и содействует отладке. Практически все типы исключений предлагают аналогичный конструктор.

Повторная генерация *менее* специфичного исключения может осуществляться при пересечении границы доверия, чтобы не допустить утечки технической информации потенциальным взломщикам.

## Основные свойства класса `System.Exception`

Ниже описаны наиболее важные свойства класса `System.Exception`.

`StackTrace`

Строка, представляющая все методы, которые были вызваны, начиная с источника исключения и заканчивая блоком `catch`.

`Message`

Строка с описанием ошибки.

`InnerException`

Внутреннее исключение (если есть), которое привело к генерации внешнего исключения. Само свойство может иметь отличающееся свойство `InnerException`.

Все исключения в C# происходят во время выполнения — эквивалент проверяемых исключений этапа компиляции Java в языке C# отсутствует.

## Общие типы исключений

Перечисленные ниже типы исключений широко используются в CLR и .NET Framework. Их можно генерировать либо применять в качестве базовых классов для порождения специальных типов исключений.

`System.ArgumentException`

Генерируется, когда функция вызывается с недопустимым аргументом. Как правило, указывает на наличие ошибки в программе.

`System.ArgumentNullException`

Подкласс `ArgumentException`, который генерируется, когда аргумент функции (непредсказуемо) равен `null`.

`System.ArgumentOutOfRangeException`

Подкласс `ArgumentException`, который генерируется, когда (обычно числовой) аргумент имеет слишком большое или слишком малое значение. Например, это исключение возникает при передаче отрицательного числа в функцию, которая принимает только положительные значения.

`System.InvalidOperationException`

Генерируется, когда состояние объекта оказывается неподходящим для успешного выполнения метода независимо от любых заданных значений аргументов. В качестве примеров можно назвать чтение неоткрытого файла или получение следующего элемента из перечислителя в случае, если лежащий в основе список был изменен на середине выполнения итерации.

`System.NotSupportedException`

Генерируется для указания на то, что специфическая функциональность не поддерживается. Хорошим примером может служить вызов метода `Add` на коллекции, для которой свойство `IsReadOnly` возвращает `true`.

`System.NotImplementedException`

Генерируется для указания на то, что функция пока еще не реализована.

`System.ObjectDisposedException`

Генерируется, когда объект, на котором вызывается функция, был освобожден.

Еще одним часто встречающимся типом исключения является `NullReferenceException`. Среда CLR генерирует такое исключение, когда вы пытаетесь получить доступ к члену объекта, значение которого равно `null` (указывая на ошибку в коде). Исключение `NullReferenceException` можно генерировать напрямую (в тестовых целях) следующим образом:

```
throw null;
```

## Шаблон методов `TryXXX`

При написании метода у вас есть выбор – возвратить код неудачи некоторого вида либо сгенерировать исключение – в ситуации, когда что-то пошло не так. В общем случае исключение следует генерировать, если ошибка находится за рамками нормального рабочего потока или ожидается, что непосредственно вызвавший код неспособен с ней справиться. Тем не менее, иногда лучше предложить потребителю оба варианта. Примером может служить тип `int`, в котором определены две версии метода `Parse`, отвечающего за разбор:

```
public int Parse     (string input);
public bool TryParse (string input, out int returnValue);
```

Если разбор заканчивается неудачей, то метод `Parse` генерирует исключение, а `TryParse` возвращает значение `false`.

Такой шаблон можно реализовать, обеспечив вызов метода `Try`*XXX* внутри метода *XXX*:

```
public возвращаемый-тип XXX (входной-тип input)
{
  возвращаемый-тип returnValue;
  if (!TryXXX (input, out returnValue))
    throw new YYYException (...)
  return returnValue;
}
```

## Альтернативы исключениям

Как и метод `int.TryParse`, функция может сообщать о неудаче путем возвращения кода ошибки вызывающей функции через возвращаемый тип или параметр. Хотя такой подход хорошо работает с простыми и предсказуемыми отказами, он становится громоздким при необходимости охвата всех ошибок, засоряя сигнатуры методов и привнося ненужную сложность и беспорядок. Кроме того, его нельзя распространить на функции, не являющиеся методами, такие как операции (например, деление) или свойства. В качестве альтернативы информация об ошибке может храниться в общем местоположении, в котором его способны видеть все функции из стека вызовов (ска-

жем, можно иметь статический метод, сохраняющий текущий признак ошибки для потока). Тем не менее, это требует от каждой функции участия в шаблоне распространения ошибок, который мало того, что громоздкий, но по иронии судьбы сам подвержен ошибкам.

# Перечисление и итераторы

## Перечисление

*Перечислитель* – это допускающий только чтение однонаправленный курсор по *последовательности значений*. Перечислитель представляет собой объект, который реализует один из двух интерфейсов:

- `System.Collections.IEnumerator`
- `System.Collections.Generic.IEnumerator<T>`

Формально любой объект, который имеет метод по имени `MoveNext` и свойство под названием `Current`, трактуется как перечислитель. Такое ослабление было введено в версии C# 1.0, чтобы избежать накладных расходов упаковки/распаковки при перечислении элементов, относящихся к типам значений, но стало избыточным после появления обобщений в C# 2.

Оператор `foreach` выполняет итерацию по *перечислимому* объекту. Перечислимый объект является логическим представлением последовательности. Это не собственно курсор, а объект, который производит курсор на себе самом. Перечислимый объект обладает одной из двух характеристик:

- реализует интерфейс `IEnumerable` или `IEnumerable<T>`;
- имеет метод по имени `GetEnumerator`, который возвращает *перечислитель*.

Интерфейсы `IEnumerator` и `IEnumerable` определены в пространстве имен `System.Collections`, а интерфейсы `IEnumerator<T>` и `IEnumerable<T>` – в пространстве имен `System.Collections.Generic`.

Шаблон перечисления выглядит следующим образом:

```
class Enumerator // Обычно реализует интерфейс IEnumerator или IEnumerator<T>
{
  public IteratorVariableType Current { get {...} }
  public bool MoveNext() {...}
}
class Enumerable // Обычно реализует интерфейс IEnumerable или IEnumerable<T>
{
  public Enumerator GetEnumerator() {...}
}
```

Ниже показан высокоуровневый способ выполнения итерации по символам в слове *beer* с использованием оператора `foreach`:

```
foreach (char c in "beer")
  Console.WriteLine (c);
```

А вот низкоуровневый метод проведения итерации по символам в слове *beer* без применения оператора `foreach`:

```
using (var enumerator = "beer".GetEnumerator())
  while (enumerator.MoveNext())
  {
    var element = enumerator.Current;
    Console.WriteLine (element);
  }
```

Если перечислитель реализует интерфейс `IDisposable`, то оператор `foreach` также действует как оператор `using`, неявно освобождая объект перечислителя.

Интерфейсы перечисления более подробно рассматриваются в главе 7.

## Инициализаторы коллекций

Перечислимый объект можно создать и заполнить за один шаг. Например:

```
using System.Collections.Generic;
...
List<int> list = new List<int> {1, 2, 3};
```

Компилятор транслирует это в следующий код:

```
using System.Collections.Generic;
...
List<int> list = new List<int>();
list.Add (1);
list.Add (2);
list.Add (3);
```

Здесь требуется, чтобы перечислимый объект реализовывал интерфейс `System.Collections.IEnumerable` и потому имел метод `Add`, который принимает соответствующее количество параметров для вызова. Похожим образом можно инициализировать словари (см. раздел "Словари" в главе 7):

```
var dict = new Dictionary<int, string>()
{
  { 5, "five" },
  { 10, "ten" }
};
```

Или более компактно:

```
var dict = new Dictionary<int, string>()
{
  [3] = "three",
  [10] = "ten"
};
```

Последний код допустим не только со словарями, но и с любым типом, для которого существует индексатор.

## Итераторы

Оператор `foreach` можно считать *потребителем* перечислителя, а итератор – *поставщиком* перечислителя. В приведенном ниже примере итератор используется для возвращения последовательности чисел Фибоначчи (где каждое число является суммой двух предыдущих чисел):

```
using System;
using System.Collections.Generic;
```

```
class Test
{
  static void Main()
  {
    foreach (int fib in Fibs(6))
      Console.Write (fib + "  ");
  }
  static IEnumerable<int> Fibs (int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }
}
```

ВЫВОД: 1 1 2 3 5 8

В то время как оператор `return` выражает: "Вот значение, которое должно быть возвращено из этого метода", оператор `yield return` сообщает: "Вот следующий элемент, который должен быть выдан этим перечислителем". С каждым оператором `yield` управление возвращается вызывающему компоненту, но состояние вызываемого метода сохраняется, так что данный метод может продолжить свое выполнение, как только вызывающий компонент перечислит следующий элемент. Жизненный цикл такого состояния ограничен перечислителем, поэтому состояние может быть освобождено, когда вызывающий компонент завершит перечисление.

Компилятор преобразует методы итератора в закрытые классы, которые реализуют интерфейсы `IEnumerable<T>` и/или `IEnumerator<T>`. Логика внутри блока итератора "инвертируется" и сращивается с методом `MoveNext` и свойством `Current` класса перечислителя, сгенерированного компилятором. Это значит, что при вызове метода итератора всего лишь создается экземпляр сгенерированного компилятором класса; никакой написанный вами код на самом деле не выполняется! Ваш код запускается только когда начинается перечисление по результирующей последовательности, обычно с помощью оператора `foreach`.

## Семантика итератора

Итератор представляет собой метод, свойство или индексатор, который содержит один или большее количество операторов `yield`. Итератор должен возвращать один из следующих четырех интерфейсов (иначе компилятор сообщит об ошибке):

```
// Перечислимые интерфейсы
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>

// Интерфейсы перечислителя
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

Семантика итератора отличается в зависимости от того, что он возвращает — реализацию *перечислимого* интерфейса или реализацию интерфейса *перечислителя* (за более подробным описанием обращайтесь в главу 7).

Разрешено применять *несколько операторов* `yield`. Например:

```
class Test
{
  static void Main()
  {
    foreach (string s in Foo())
      Console.WriteLine(s);  // Выводит "One","Two","Three"
  }

  static IEnumerable<string> Foo()
  {
    yield return "One";
    yield return "Two";
    yield return "Three";
  }
}
```

### Оператор `yield break`

Оператор `yield break` указывает, что блок итератора должен быть завершен преждевременно, не возвращая больше элементов. Для его демонстрации модифицируем метод `Foo`, как показано ниже:

```
static IEnumerable<string> Foo (bool breakEarly)
{
  yield return "One";
  yield return "Two";

  if (breakEarly)
    yield break;

  yield return "Three";
}
```

Наличие оператора `return` в блоке итератора не допускается — вместо него должен использоваться `yield break`.

### Итераторы и блоки `try/catch/finally`

Оператор `yield return` не может присутствовать в блоке `try`, который имеет конструкцию `catch`:

```
IEnumerable<string> Foo()
{
  try { yield return "One"; }  // Не допускается
  catch { ... }
}
```

Также оператор `yield return` нельзя применять внутри блока `catch` или `finally`. Указанные ограничения объясняются тем фактом, что компилятор должен транслировать итераторы в обычные классы с членами `MoveNext`, `Current` и `Dispose`, а трансляция блоков обработки исключений может привести к чрезмерной сложности.

Однако оператор `yield` можно использовать в блоке `try`, который имеет (только) блок `finally`:

```
IEnumerable<string> Foo()
{
```

```
  try { yield return "One"; }  // Нормально
  finally { ... }
}
```

Код в блоке `finally` выполняется, когда потребляемый перечислитель достигает конца последовательности или освобождается. Оператор `foreach` неявно освобождает перечислитель, если произошло преждевременное завершение, обеспечивая безопасный способ применения перечислителей. При явной работе с перечислителем частой ловушкой оказывается преждевременное прекращение перечисления без освобождения перечислителя, т.е. в обход блока `finally`. Во избежание подобного риска код, явно использующий итератор, можно поместить внутрь оператора `using`:

```
string firstElement = null;
var sequence = Foo();
using (var enumerator = sequence.GetEnumerator())
  if (enumerator.MoveNext())
    firstElement = enumerator.Current;
```

## Компоновка последовательностей

Итераторы в высшей степени компонуемы. Мы можем расширить наш пример с числами Фибоначчи, выводя только четные числа Фибоначчи:

```
using System;
using System.Collections.Generic;

class Test
{
  static void Main()
  {
    foreach (int fib in EvenNumbersOnly (Fibs(6)))
      Console.WriteLine (fib);
  }

  static IEnumerable<int> Fibs (int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }

  static IEnumerable<int> EvenNumbersOnly (IEnumerable<int> sequence)
  {
    foreach (int x in sequence)
      if ((x % 2) == 0)
        yield return x;
  }
}
```

Каждый элемент не вычисляется вплоть до последнего момента – когда он запрашивается операцией `MoveNext`. На рис. 4.1 показаны запросы данных и их вывод с течением времени.

Возможность компоновки, поддерживаемая шаблоном итератора, жизненно необходима при построении запросов LINQ; мы подробно обсудим данную тему в главе 8.

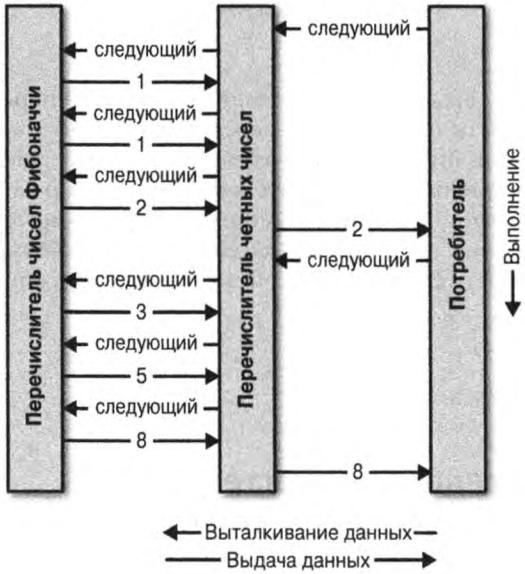



**Рис. 4.1.** *Пример компоновки последовательностей*

# Типы, допускающие значение `null`

Ссылочные типы могут представлять несуществующее значение с помощью ссылки `null`. Тем не менее, типы значений не способны представлять значения `null` обычным образом. Например:

```
string s = null;  // Нормально, ссылочный тип
int i = null;     // Ошибка на этапе компиляции, тип int не может быть null
```

Чтобы представить `null` с помощью типа значения, нужно применять специальную конструкцию, которая называется *типом, допускающим значение* `null`. Тип, допускающий значение `null`, обозначается как тип значения, за которым следует символ `?`:

```
int? i = null;                  // Нормально; тип, допускающий значение null
Console.WriteLine (i == null);  // True
```

## Структура `Nullable<T>`

Тип `T?` транслируется в `System.Nullable<T>`. Тип `Nullable<T>` является легковесной неизменяемой структурой, которая имеет только два поля, предназначенные для представления значения (`Value`) и признака наличия значения (`HasValue`). По существу структура `System.Nullable<T>` очень проста:

```
public struct Nullable<T> where T : struct
{
  public T Value {get;}
  public bool HasValue {get;}
  public T GetValueOrDefault();
  public T GetValueOrDefault (T defaultValue);
  ...
}
```

Код:

```
int? i = null;
Console.WriteLine (i == null);     // True
```

транслируется в:

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue); // True
```

Попытка извлечь значение Value, когда в поле HasValue содержится false, приводит к генерации исключения InvalidOperationException. Метод GetValueOrDefault возвращает значение Value, если HasValue равно true, и результат new T() или заданное стандартное значение в противном случае.

Стандартным значением T? является null.

### Неявные и явные преобразования с участием типов, допускающих значение null

Преобразование из T в T? будет неявным, а из T? в T — явным. Например:

```
int? x = 5;          // неявное
int y = (int)x;      // явное
```

Явное приведение полностью эквивалентно обращению к свойству Value объекта типа, допускающего null. Следовательно, если HasValue равно false, тогда генерируется исключение InvalidOperationException.

### Упаковка и распаковка значений типов, допускающих null

Когда T? упаковывается, упакованное значение в куче содержит T, а не T?. Такая оптимизация возможна из-за того, что упакованное значение относится к ссылочному типу, который уже способен выражать null. В C# также разрешено распаковывать типы, допускающие null, с помощью операции as. Если приведение не удается, тогда результатом будет null:

```
object o = "string";
int? x = o as int?;
Console.WriteLine (x.HasValue);  // False
```

## Подъем операций

В структуре Nullable<T> не определены такие операции, как <, > или даже ==. Несмотря на это, следующий код успешно компилируется и выполняется:

```
int? x = 5;
int? y = 10;
bool b = x < y;  // true
```

Код работает благодаря тому, что компилятор заимствует, или "поднимает", операцию "меньше чем" у лежащего в основе типа значения. Семантически предыдущее выражение сравнения транслируется так:

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

Другими словами, если x и y имеют значения, то сравнение производится посредством операции "меньше чем" типа int; в противном случае результатом будет false.

Подъем операций означает возможность неявного использования операций из T для типа T?. Вы можете определить операции для T?, чтобы предоставить специализированное поведение в отношении null, но в подавляющем большинстве случаев лучше полагаться на автоматическое применение компилятором систематической логики работы со значением null.

Ниже показано несколько примеров:

```
int? x = 5;
int? y = null;

// Примеры использования операции эквивалентности
Console.WriteLine (x == y);        // False
Console.WriteLine (x == null);     // False
Console.WriteLine (x == 5);        // True
Console.WriteLine (y == null);     // True
Console.WriteLine (y == 5);        // False
Console.WriteLine (y != 5);        // True

// Примеры применения операций отношения
Console.WriteLine (x < 6);         // True
Console.WriteLine (y < 6);         // False
Console.WriteLine (y > 6);         // False

// Примеры использования всех других операций
Console.WriteLine (x + 5);         // 10
Console.WriteLine (x + y);         // null (выводит пустую строку)
```

Компилятор представляет логику в отношении `null` по-разному в зависимости от категории операции. Правила объясняются в последующих разделах.

### Операции эквивалентности (== и !=)

Поднятые операции эквивалентности обрабатывают значения `null` точно так же, как поступают ссылочные типы. Это означает, что два значения `null` равны:

```
Console.WriteLine (       null ==       null);   // True
Console.WriteLine ((bool?)null == (bool?)null); // True
```

Более того:

- если в точности один операнд имеет значение `null`, то операнды не равны;
- если оба операнда отличны от `null`, то сравниваются их свойства `Value`.

### Операции отношения (<, <=, >=, >)

Работа операций отношения основана на принципе, согласно которому сравнение операндов `null` не имеет смысла. Это означает, что сравнение `null` либо с `null`, либо со значением, отличным от `null`, дает в результате `false`:

```
bool b = x < y;  // Транслируется в:

bool b = (x.HasValue && y.HasValue)
       ? (x.Value < y.Value)
       : false;

// b равно false (предполагая, что x равно 5, а y - null)
```

### Остальные операции (+, –, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~)

Остальные операции возвращают `null`, когда любой из операндов равен `null`. Такой шаблон должен быть хорошо знакомым пользователям языка SQL:

```
int? c = x + y;  // Транслируется в:

int? c = (x.HasValue && y.HasValue)
       ? (int?) (x.Value + y.Value)
       : null;

// c равно null (предполагая, что x равно 5, а y - null)
```

Исключением будет ситуация, когда операции & и | применяются к bool?, что мы вскоре обсудим.

### Смешивание типов, допускающих и не допускающих null

Типы, допускающие и не допускающие null, можно смешивать (прием работает, поскольку существует неявное преобразование из T в T?):

```
int? a = null;
int b = 2;
int? c = a + b;  // c равно null - эквивалентно a + (int?)b
```

## Тип bool? и операции & и |

Когда предоставляются операнды типа bool?, операции & и | трактуют null как *неизвестное значение*. Таким образом, null | true дает true по следующим причинам:

- если неизвестное значение равно false, то результатом будет true;
- если неизвестное значение равно true, то результатом будет true.

Аналогичным образом null & false дает false. Подобное поведение должно быть знакомым пользователям языка SQL. Ниже приведены другие комбинации:

```
bool? n = null;
bool? f = false;
bool? t = true;
Console.WriteLine (n | n);  // (null)
Console.WriteLine (n | f);  // (null)
Console.WriteLine (n | t);  // True
Console.WriteLine (n & n);  // (null)
Console.WriteLine (n & f);  // False
Console.WriteLine (n & t);  // (null)
```

## Типы, допускающие null, и операции для работы со значениями null

Типы, допускающие значение null, особенно хорошо работают с операцией ?? (см. раздел "Операция объединения с null" в главе 2). Например:

```
int? x = null;
int y = x ?? 5;  // y равно 5

int? a = null, b = 1, c = 2;
Console.WriteLine (a ?? b ?? c);  // 1 (первое значение, отличное от null)
```

Использование операции ?? эквивалентно вызову метода GetValueOrDefault с явным стандартным значением за исключением того, что выражение для стандартного значения никогда не оценивается, если переменная не равна null.

Типы, допускающие значение null, также удобно применять с null-условной операцией (см. раздел "null-условная операция (C# 6)" в главе 2). В следующем примере переменная length получает значение null:

```
System.Text.StringBuilder sb = null;
int? length = sb?.ToString().Length;
```

Скомбинировав этот код с операцией объединения с null, переменной length можно присвоить значение 0 вместо null:

```
int length = sb?.ToString().Length ?? 0;     // length получает значение 0,
                                             // если sb равно null
```

## Сценарии использования типов, допускающих null

Один из самых распространенных сценариев использования типов, допускающих null – представление неизвестных значений. Он часто встречается в программировании для баз данных, когда класс отображается на таблицу со столбцами, допускающими значение null. Если такие столбцы хранят строковые значения (например, столбец EmailAddress в таблице Customer), то проблемы не возникают, потому что строка в среде CLR является ссылочным типом, который может быть null. Однако большинство других типов столбцов SQL отображаются на типы структур CLR, что делает типы, допускающие null, очень удобными при отображении типов SQL на типы CLR. Например:

```
// Отображается на таблицу Customer в базе данных
public class Customer
{
  ...
  public decimal? AccountBalance;
}
```

Тип, допускающий null, может также применяться для представления поддерживающего поля в так называемом *свойстве окружения* (ambient property). Когда свойство окружения равно null, оно возвращает значение своего родителя. Например:

```
public class Row
{
  ...
  Grid parent;
  Color? color;
  public Color Color
  {
    get { return color ?? parent.Color; }
    set { color = value == parent.Color ? (Color?)null : value; }
  }
}
```

## Альтернативы типам, допускающим значение null

До того, как типы, допускающие null, стали частью языка C# (т.е. до версии C# 2.0), существовало много стратегий для работы с типами значений, допускающими null, и по историческим причинам их примеры по-прежнему можно встретить в .NET Framework. Одна из таких стратегий состояла в том, что какое-то конкретное отличное от null значение определялось как "значение null"; примеры можно найти в классах строк и массивов. Метод String.IndexOf возвращает "магическое значение" -1, когда символ в строке не обнаружен:

```
int i = "Pink".IndexOf ('b');
Console.WriteLine (i);  // -1
```

Тем не менее, метод Array.IndexOf возвращает -1, только если индекс ограничен 0. Более общий рецепт заключается в том, что IndexOf возвращает значение на единицу меньше нижней границы массива. В следующем примере IndexOf возвращает 0, если элемент не найден:

```
// Создать массив, нижняя граница которого равна 1, а не 0:
Array a = Array.CreateInstance (typeof (string),
                                new int[] {2}, new int[] {1});
a.SetValue ("a", 1);
```

```
a.SetValue ("b", 2);
Console.WriteLine (Array.IndexOf (a, "c"));  // 0
```

Выбор "магического значения" сопряжен с проблемами по нескольким причинам.

- Такой подход приводит к тому, что каждый тип значения имеет разное представление `null`. В противоположность этому типы, допускающие `null`, предлагают один общий шаблон, который работает для всех типов значений.
- Приемлемого значения может и не быть. В предыдущем примере всегда использовать `-1` не получится. То же самое справедливо для ранее приведенного примера, представляющего неизвестный баланс счета (`AccountBalance`).
- Если забыть о проверке на равенство "магическому значению", то появится некорректное значение, которое может оказаться незамеченным вплоть до этапа выполнения – когда возникнет неожиданное поведение. С другой стороны, если забыть о проверке `HasValue` на равенство `null`, тогда немедленно сгенерируется исключение `InvalidOperationException`.
- Способность значения быть `null` не отражена в *типе*. Типы сообщают о целях программы, позволяя компилятору проверять корректность и применять согласованный набор правил.

# Расширяющие методы

*Расширяющие методы* позволяют расширять существующий тип новыми методами, не изменяя определение исходного типа. Расширяющий метод – это статический метод статического класса, в котором к первому параметру применен модификатор `this`. Типом первого параметра должен быть тип, который расширяется. Например:

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
  }
}
```

Расширяющий метод `IsCapitalized` может вызываться так, как если бы он был методом экземпляра класса `string`:

```
Console.WriteLine ("Perth".IsCapitalized());
```

Вызов расширяющего метода при компиляции транслируется в обычный вызов статического метода:

```
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```

Такая трансляция работает следующим образом:

```
arg0.Method (arg1, arg2, ...);              // Вызов расширяющего метода
StaticClass.Method (arg0, arg1, arg2, ...); // Вызов статического метода
```

Интерфейсы также можно расширять:

```
public static T First<T> (this IEnumerable<T> sequence)
{
  foreach (T element in sequence)
    return element;
```

```
    throw new InvalidOperationException ("No elements!"); // элементы отсутствуют
  }
  ...
  Console.WriteLine ("Seattle".First());   // S
```

Расширяющие методы появились в версии C# 3.0.

## Цепочки расширяющих методов

Как и методы экземпляра, расширяющие методы предлагают аккуратный способ для связывания функций в цепочки. Взгляните на следующие две функции:

```
public static class StringHelper
{
  public static string Pluralize  (this string s) {...}
  public static string Capitalize (this string s) {...}
}
```

Строковые переменные x и y эквивалентны и получают значение "Sausages", но x использует расширяющие методы, тогда как y — статические:

```
string x = "sausage".Pluralize().Capitalize();
string y = StringHelper.Capitalize (StringHelper.Pluralize ("sausage"));
```

## Неоднозначность и разрешение

### Пространства имен

Расширяющий метод не может быть доступен до тех пор, пока его пространство имен не окажется в области видимости, обычно за счет импорта посредством оператора using. Взгляните на приведенный далее расширяющий метод IsCapitalized:

```
using System;

namespace Utils
{
  public static class StringHelper
  {
    public static bool IsCapitalized (this string s)
    {
      if (string.IsNullOrEmpty(s)) return false;
      return char.IsUpper (s[0]);
    }
  }
}
```

Для применения метода IsCapitalized в показанном ниже приложении должно импортироваться пространство имен Utils, иначе на этапе компиляции возникнет ошибка:

```
namespace MyApp
{
  using Utils;

  class Test
  {
    static void Main() => Console.WriteLine ("Perth".IsCapitalized());
  }
}
```

### Расширяющий метод или метод экземпляра

Любой совместимый метод экземпляра всегда будет иметь преимущество над расширяющим методом. В следующем примере предпочтение всегда будет отдаваться методу `Foo` класса `Test` – даже если осуществляется вызов с аргументом `x` типа `int`:

```
class Test
{
  public void Foo (object x) { }  // Этот метод всегда имеет преимущество
}

static class Extensions
{
  public static void Foo (this Test t, int x) { }
}
```

Единственный способ обратиться к расширяющему методу в такой ситуации – воспользоваться нормальным статическим синтаксисом; другими словами, `Extensions.Foo(...)`.

### Расширяющий метод или другой расширяющий метод

Если два расширяющих метода имеют одинаковые сигнатуры, то расширяющий метод должен вызываться как обычный статический метод, чтобы устранить неоднозначность при вызове. Однако если один расширяющий метод имеет более специфичные аргументы, тогда ему будет отдаваться предпочтение.

Для иллюстрации сказанного рассмотрим два класса:

```
static class StringHelper
{
  public static bool IsCapitalized (this string s) {...}
}

static class ObjectHelper
{
  public static bool IsCapitalized (this object s) {...}
}
```

В следующем коде вызывается метод `IsCapitalized` класса `StringHelper`:

```
bool test1 = "Perth".IsCapitalized();
```

Классы и структуры считаются более специфичными, чем интерфейсы.

## Анонимные типы

Анонимный тип – это простой класс, созданный на лету с целью хранения набора значений. Для создания анонимного типа применяется ключевое слово `new` с инициализатором объекта, указывающим свойства и значения, которые будет содержать тип. Например:

```
var dude = new { Name = "Bob", Age = 23 };
```

Компилятор транслирует данный оператор в (приблизительно) такой код:

```
internal class AnonymousGeneratedTypeName
{
  private string name;    // Фактическое имя поля несущественно
  private int    age;     // Фактическое имя поля несущественно
```

```
  public AnonymousGeneratedTypeName (string name, int age)
  {
    this.name = name; this.age = age;
  }
  public string  Name { get { return name; } }
  public int     Age  { get { return age;  } }
  // Методы Equals и GetHashCode переопределены (см. главу 6).
  // Метод ToString также переопределен.
}
...
var dude = new AnonymousGeneratedTypeName ("Bob", 23);
```

При ссылке на анонимный тип должно использоваться ключевое слово `var`, т.к. имя этого типа не известно.

Имя свойства анонимного типа может быть выведено из выражения, которое само по себе является идентификатором (или завершается им). Например:

```
int Age = 23;
var dude = new { Name = "Bob", Age, Age.ToString().Length };
```

эквивалентно:

```
var dude = new { Name = "Bob", Age = Age, Length = Age.ToString().Length };
```

Два экземпляра анонимного типа, объявленные внутри одной сборки, будут иметь один и тот же лежащий в основе тип, если их элементы именованы и типизированы идентичным образом:

```
var a1 = new { X = 2, Y = 4 };
var a2 = new { X = 2, Y = 4 };
Console.WriteLine (a1.GetType() == a2.GetType());  // True
```

Кроме того, метод `Equals` переопределен, чтобы выполнять сравнения эквивалентности:

```
Console.WriteLine (a1 == a2);          // False
Console.WriteLine (a1.Equals (a2));    // True
```

Можно создавать массивы анонимных типов, как показано ниже:

```
var dudes = new[]
{
  new { Name = "Bob", Age = 30 },
  new { Name = "Tom", Age = 40 }
};
```

Метод не может (удобно) возвращать объект анонимного типа, поскольку писать метод с возвращаемым типом `var` не допускается:

```
var Foo() => new { Name = "Bob", Age = 30 };  // Незаконно!
```

Взамен потребуется применить `object` или `dynamic` и тогда код, вызывающий метод `Foo`, обязан полагаться на динамическое связывание, утрачивая статическую безопасность типов (и средство IntelliSense в Visual Studio):

```
dynamic Foo() => new { Name = "Bob", Age = 30 };  // Статическая безопасность
                                                  // типов отсутствует
```

Анонимные типы используются главным образом при написании запросов LINQ (глава 8) и появились в версии C# 3.0.

# Кортежи (C# 7)

Подобно анонимным типам кортежи предлагают простой способ хранения набора значений. Главный замысел кортежей — безопасно возвращать множество значений из метода, не прибегая к параметрам `out` (то, что невозможно делать с помощью анонимных типов).

Кортежи в C# 7 поддерживают почти все, что обеспечивают анонимные типы, и даже больше. Как вскоре вы увидите, их основной недостаток связан со стиранием типов во время выполнения вместе с именованными элементами.

Создать *литеральный кортеж* проще всего, указав в круглых скобках список желаемых значений. В результате создается кортеж с *неименованными* элементами, на которые можно ссылаться как на `Item1`, `Item2` и т.д.:

```
var bob = ("Bob", 23);  // Позволить компилятору вывести типы элементов
Console.WriteLine (bob.Item1);  // Bob
Console.WriteLine (bob.Item2);  // 23
```

Функциональность кортежей C# 7 полагается на набор поддерживающих обобщенных структур по имени `System.ValueTuple<...>`. Они не являются частью .NET Framework 4.6 и содержатся в сборке под названием `System.ValueTuple`, которая доступна в пакете NuGet с тем же самым именем. Если вы работаете с Visual Studio и .NET Framework 4.6, тогда должны явно загрузить упомянутый пакет. (Если вы используете LINQPad, то требуемая сборка включается автоматически.)

В версии .NET Framework 4.7 сборка `System.ValueTuple` встроена в `mscorlib.dll`.

Кортежи являются *типами значений* с *изменяемыми* (допускающими чтение/запись) элементами:

```
var joe = bob;           // joe - *копия* bob
joe.Item1 = "Joe";       // Изменить значение элемента Item1 в joe с Bob на Joe
Console.WriteLine (bob); // (Bob, 23)
Console.WriteLine (joe); // (Joe, 23)
```

В отличие от анонимных типов *тип кортежа* можно указывать явно. Нужно лишь перечислить типы элементов в круглых скобках:

```
(string,int) bob = ("Bob", 23); //Ключевое слово var для кортежей не обязательно!
```

Это означает, что кортеж можно удобно возвращать из метода:

```
static (string,int) GetPerson() => ("Bob", 23);

static void Main()
{
  (string,int) person = GetPerson();  // При желании здесь можно было бы
                                      // применить var
  Console.WriteLine (person.Item1);   // Bob
  Console.WriteLine (person.Item2);   // 23
}
```

Кортежи хорошо сочетаются с обобщениями, так что все следующие типы законны:

```
Task<(string,int)>
Dictionary<(string,int),Uri>
IEnumerable<(int ID, string Name)>  // См. ниже...
```

## Именование элементов кортежа

При создании литеральных кортежей элементам можно дополнительно назначать содержательные имена:

```
var tuple = (Name:"Bob", Age:23);

Console.WriteLine (tuple.Name);    // Bob
Console.WriteLine (tuple.Age);     // 23
```

То же самое разрешено делать при указании *типов кортежей*:

```
static (string Name, int Age) GetPerson() => ("Bob", 23);

static void Main()
{
  var person = GetPerson();
  Console.WriteLine (person.Name); // Bob
  Console.WriteLine (person.Age);  // 23
}
```

Обратите внимание, что элементы по-прежнему можно трактовать как неименованные и ссылаться на них как на `Item1`, `Item2` и т.д. (хотя Visual Studio скрывает эти поля от IntelliSense).

Кортежи совместимы по типу друг с другом, если типы их элементов совпадают (по порядку). Совпадение имен элементов не обязательно:

```
(string Name, int Age, char Sex) bob1 = ("Bob", 23, 'M');
(string Age, int Sex, char Name) bob2 = bob1;  // Ошибки нет!
```

Рассматриваемый пример приводит к сбивающим с толку результатам:

```
Console.WriteLine (bob2.Name);     // M
Console.WriteLine (bob2.Age);      // Bob
Console.WriteLine (bob2.Sex);      // 23
```

### Стирание типов

Ранее мы заявляли, что компилятор C# поддерживает анонимные типы путем построения специальных классов с именованными свойствами для каждого элемента. В случае кортежей компилятор C# работает по-другому и задействует существующее семейство обобщенных структур:

```
public struct ValueTuple<T1>
public struct ValueTuple<T1,T2>
public struct ValueTuple<T1,T2,T3>
...
```

Каждая структура `ValueType<>` содержит поля с именами `Item1`, `Item2` и т.д.

Следовательно, `(string,int)` является псевдонимом для `ValueTuple <string,int>`, а это значит, что именованные элементы кортежей не имеют соответствующих имен свойств внутри лежащих в основе типов. Взамен имена существуют только в исходном коде и в воображении компилятора. Во время выполнения имена по обыкновению исчезают, так что после декомпиляции программы, которая ссылается на именованные элементы кортежей, вы увидите только ссылки на `Item1`, `Item2`

и т.д. Более того, когда вы исследуете переменную типа кортежа в отладчике после ее присваивания экземпляру `object` (или выводите ее на экран в LINQPad), имена элементов отсутствуют. И самое главное — вы не можете использовать *рефлексию* (глава 19) для определения имен элементов кортежа во время выполнения.

Мы говорим *по обыкновению* исчезают, потому что есть исключение. Для методов/свойств, которые возвращают именованные типы кортежей, компилятор выпускает имена элементов, применяя к возвращаемому типу члена специальный атрибут под названием `TupleElementNamesAttribute` (см. раздел "Атрибуты" далее в главе). Это позволяет именованным элементам работать при вызове методов в другой сборке (исходный код которой компилятору не доступен).

## Метод `ValueTuple.Create`

Кортежи можно создавать также через фабричный метод из (необобщенного) типа `ValueTuple`:

```
ValueTuple<string,int> bob1 = ValueTuple.Create ("Bob", 23);
(string,int)           bob2 = ValueTuple.Create ("Bob", 23);
```

Таким способом не могут создаваться именованные элементы, поскольку именованием элементов занимается компилятор.

## Деконструирование кортежей

Кортежи неявно поддерживают шаблон деконструирования (см. раздел "Деконструкторы (C# 7)" в главе 1), поэтому кортеж можно легко *деконструировать* в индивидуальные переменные. Следовательно, вместо кода:

```
var bob = ("Bob", 23);
string name = bob.Item1;
int age = bob.Item2;
```

можно написать:

```
var bob = ("Bob", 23);
(string name, int age) = bob; // Деконструировать кортеж bob в
                              // отдельные переменные (name и age).
Console.WriteLine (name);
Console.WriteLine (age);
```

Синтаксис деконструирования очень похож на синтаксис объявления кортежа с именованными элементами, что может привести к путанице! Разница подчеркивается в приведенном ниже коде:

```
(string name, int age) = bob;       // Деконструирование кортежа
(string name, int age) bob2 = bob; // Объявление нового кортежа
```

Вот еще один пример, на этот раз с вызовом метода и выведением типа (`var`):

```
static (string, int, char) GetBob() => ( "Bob", 23, 'M');
static void Main()
{
  var (name, age, sex) = GetBob();
  Console.WriteLine (name);   // Bob
  Console.WriteLine (age);    // 23
  Console.WriteLine (sex);    // M
}
```

## Сравнение эквивалентности

Как и анонимные типы, типы `ValueTuple<>` переопределяют метод `Equals`, чтобы обеспечить содержательную работу сравнений эквивалентности:

```
var t1 = ("one", 1);
var t2 = ("one", 1);
Console.WriteLine (t1.Equals (t2));  // True
```

В результате кортежи удобно использовать в качестве ключей в словарях. Сравнение эквивалентности подробно рассматривается в главе 6, а словари – в главе 7.

Типы `ValueTuple<>` также реализуют интерфейс `IComparable` (см. раздел "Сравнение порядка" в главе 6), делая возможным применение кортежей как ключей сортировки.

## Классы `System.Tuple`

В пространстве имен `System` вы обнаружите еще одно семейство обобщенных типов под названием `Tuple` (не `ValueTuple`). Они появились в .NET Framework 4.0 и являются классами (тогда как типы `ValueTuple` представляют собой структуры). Оглядываясь назад, можно сказать, что определение кортежей как классов было заблуждением: в типовых сценариях, где используются кортежи, структуры обеспечивают небольшое преимущество в плане производительности (за счет избегания излишних выделений памяти), практически не имея недостатков. Поэтому при добавлении в C# 7 языковой поддержки для кортежей существующие типы `Tuple` были проигнорированы в пользу новых типов `ValueTuple`. Вы по-прежнему можете встречать классы `Tuple` в коде, написанном до выхода C# 7. Они не располагают специальной языковой поддержкой и применяются следующим образом:

```
Tuple<string,int> t = Tuple.Create ("Bob", 23);  // Фабричный метод
Console.WriteLine (t.Item1);  // Bob
Console.WriteLine (t.Item2);  // 23
```

# Атрибуты

Вам уже знакомо понятие снабжения элементов кода признаками в форме модификаторов, таких как `virtual` или `ref`. Эти конструкции встроены в язык. *Атрибуты* представляют собой расширяемый механизм для добавления специальной информации к элементам кода (сборкам, типам, членам, возвращаемым значениям и параметрам обобщенных типов). Такая расширяемость удобна для служб, глубоко интегрированных в систему типов, и не требует специальных ключевых слов или конструкций в языке C#.

Хороший сценарий для атрибутов связан с *сериализацией* – процессом преобразования произвольных объектов в и из определенного формата. В данном случае атрибут на поле может указывать трансляцию между представлением поля в C# и его представлением в используемом формате.

## Классы атрибутов

Атрибут определяется классом, который унаследован (прямо или косвенно) от абстрактного класса `System.Attribute`. Чтобы присоединить атрибут к элементу кода, перед элементом необходимо указать имя типа атрибута в квадратных скобках.

Например, в показанном ниже коде атрибут ObsoleteAttribute присоединяется к классу Foo:

```
[ObsoleteAttribute]
public class Foo {...}
```

Данный атрибут распознается компилятором и приводит к тому, что компилятор выдаст предупреждение, если производится ссылка на тип или член, помеченный как устаревший (obsolete). По соглашению имена всех типов атрибутов заканчиваются на *Attribute*. Такое соглашение поддерживается компилятором C# и позволяет опускать суффикс Attribute, когда присоединяется атрибут:

```
[Obsolete]
public class Foo {...}
```

Тип ObsoleteAttribute объявлен в пространстве имен System следующим образом (для краткости код упрощен):

```
public sealed class ObsoleteAttribute : Attribute {...}
```

Язык C# и инфраструктура .NET Framework включают множество предопределенных атрибутов. В главе 19 мы объясним, как создавать собственные атрибуты.

## Именованные и позиционные параметры атрибутов

Атрибуты могут иметь параметры. В показанном далее примере мы применяем к классу атрибут XmlElementAttribute, который сообщает сериализатору XML (из пространства имен System.Xml.Serialization) о том, как объект представлен в XML, и что он принимает несколько *параметров атрибута*. Таким образом, атрибут отображает класс CustomerEntity на XML-элемент по имени Customer, принадлежащий пространству имен http://oreilly.com:

```
[XmlElement ("Customer", Namespace="http://oreilly.com")]
public class CustomerEntity { ... }
```

Параметры атрибутов относятся к одной из двух категорий: позиционные и именованные. В предыдущем примере первый аргумент является *позиционным параметром*, а второй — *именованным параметром*. Позиционные параметры соответствуют параметрам открытых конструкторов типа атрибута. Именованные параметры соответствуют открытым полям или открытым свойствам типа атрибута.

При указании атрибута должны включаться позиционные параметры, которые соответствуют одному из конструкторов класса атрибута. Именованные параметры необязательны.

В главе 19 будут описаны допустимые типы параметров и правила, используемые для их проверки.

## Цели атрибутов

Неявно целью атрибута является элемент кода, находящийся непосредственно за атрибутом, который обычно представляет собой тип или член типа. Тем не менее, атрибуты можно присоединять и к сборке. При этом требуется явно указывать цель атрибута.

Вот как с помощью атрибута CLSCompliant задать соответствие общеязыковой спецификации (Common Language Specification — CLS) для целой сборки:

```
[assembly:CLSCompliant(true)]
```

## Указание нескольких атрибутов

Для одного элемента кода допускается указывать несколько атрибутов. Атрибуты могут быть заданы либо внутри единственной пары квадратных скобок (и разделяться запятыми), либо в отдельных парах квадратных скобок (или с помощью комбинации двух способов). Следующие три примера семантически идентичны:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}

[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}

[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar {...}
```

# Атрибуты информации о вызывающем компоненте

Начиная с версии C# 5.0, необязательные параметры можно помечать одним из трех *атрибутов информации о вызывающем компоненте*, которые инструктируют компилятор о необходимости передачи информации, полученной из исходного кода вызывающего компонента, в стандартное значение параметра:

- `[CallerMemberName]` применяет имя члена вызывающего компонента;
- `[CallerFilePath]` применяет путь к файлу исходного кода вызывающего компонента;
- `[CallerLineNumber]` применяет номер строки в файле исходного кода вызывающего компонента.

В приведенном далее методе `Foo` демонстрируется использование всех трех атрибутов:

```
using System;
using System.Runtime.CompilerServices;

class Program
{
  static void Main() => Foo();
  static void Foo (
    [CallerMemberName] string memberName = null,
    [CallerFilePath] string filePath = null,
    [CallerLineNumber] int lineNumber = 0)
  {
    Console.WriteLine (memberName);
    Console.WriteLine (filePath);
    Console.WriteLine (lineNumber);
  }
}
```

Предполагая, что код находится в файле `c:\source\test\Program.cs`, вывод будет таким:

```
Main
c:\source\test\Program.cs
6
```

Как и со стандартными необязательными параметрами, подстановка делается в *месте вызова*. Следовательно, показанный выше метод `Main` является "синтаксическим сахаром" для такого кода:

```
static void Main() => Foo ("Main", @"c:\source\test\Program.cs", 6);
```

Атрибуты информации о вызывающем компоненте удобны при написании функций регистрации в журнале, а также при реализации шаблонов, подобных инициированию одиночного события уведомления об изменении всякий раз, когда модифицируется любое свойство объекта. На самом деле для этого в .NET Framework предусмотрен стандартный интерфейс по имени `INotifyPropertyChanged` (из пространства имен `System.ComponentModel`):

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}

public delegate void PropertyChangedEventHandler
  (object sender, PropertyChangedEventArgs e);

public class PropertyChangedEventArgs : EventArgs
{
  public PropertyChangedEventArgs (string propertyName);
  public virtual string PropertyName { get; }
}
```

Обратите внимание, что конструктор класса `PropertyChangedEventArgs` требует имени свойства, значение которого изменяется. Однако за счет применения атрибута `[CallerMemberName]` мы можем реализовать данный интерфейс и вызвать событие, даже не указывая имена свойств:

```
public class Foo : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged = delegate { };

  void RaisePropertyChanged ([CallerMemberName] string propertyName = null)
  {
    PropertyChanged (this, new PropertyChangedEventArgs (propertyName));
  }

  string customerName;
  public string CustomerName
  {
    get { return customerName; }
    set
    {
      if (value == customerName) return;
      customerName = value;
      RaisePropertyChanged();
      // Компилятор преобразует предыдущую строку в:
      // RaisePropertyChanged ("CustomerName");
    }
  }
}
```

# Динамическое связывание

*Динамическое связывание* откладывает *связывание* — процесс распознавания типов, членов и операций — с этапа компиляции до времени выполнения. Динамическое связывание удобно, когда на этапе компиляции *вам* известно, что определенная функция, член или операция существует, но *компилятор* об этом ничего не знает. Обычно подобное происходит при взаимодействии с динамическими языками (такими как IronPython) и COM, а также в сценариях, в которых иначе использовалась бы рефлексия.

Динамический тип объявляется с помощью контекстного ключевого слова `dynamic`:

```
dynamic d = GetSomeObject();
d.Quack();
```

Динамический тип предлагает компилятору смягчить требования. Мы ожидаем, что тип времени выполнения `d` должен иметь метод `Quack`. Мы просто не можем проверить это статически. Поскольку `d` относится к динамическому типу, компилятор откладывает связывание `Quack` с `d` до времени выполнения. Понимание смысла такого действия требует уяснения различий между *статическим связыванием* и *динамическим связыванием*.

## Сравнение статического и динамического связывания

Каноническим примером связывания является отображение имени на специфическую функцию при компиляции выражения. Чтобы скомпилировать следующее выражение, компилятор должен найти реализацию метода по имени `Quack`:

```
d.Quack();
```

Давайте предположим, что статическим типом `d` является `Duck`:

```
Duck d = ...
d.Quack();
```

В простейшем случае компилятор осуществляет связывание за счет поиска в типе `Duck` метода без параметров по имени `Quack`. Если найти такой метод не удалось, тогда компилятор распространяет поиск на методы, принимающие необязательные параметры, методы базовых классов `Duck` и расширяющие методы, которые принимают тип `Duck` в своем первом параметре. Если совпадений не обнаружено, то возникает ошибка компиляции. Независимо от того, к какому методу произведено связывание, суть в том, что связывание делается компилятором, и оно полностью зависит от статических сведений о типах операндов (в данном случае `d`). Именно потому описанный процесс называется *статическим связыванием*.

А теперь изменим статический тип `d` на `object`:

```
object d = ...
d.Quack();
```

Вызов `Quack` приводит к ошибке на этапе компиляции, т.к. несмотря на то, что хранящееся в `d` значение способно содержать метод по имени `Quack`, компилятор не может об этом знать, поскольку единственная информация, которой он располагает — тип переменной, которым в рассматриваемом случае является `object`. Но давайте изменим статический тип `d` на `dynamic`:

```
dynamic d = ...
d.Quack();
```

Тип dynamic похож на object – он в равной степени не описывает тип. Отличие заключается в том, что тип dynamic допускает применение способами, которые на этапе компиляции не известны. Динамический объект связывается во время выполнения на основе своего типа времени выполнения, а не типа при компиляции. Когда компилятор встречает динамически связываемое выражение (которое в общем случае представляет собой выражение, содержащее любое значение типа dynamic), он просто упаковывает его так, чтобы связывание могло быть произведено позже во время выполнения.

Если динамический объект реализует IDynamicMetaObjectProvider, то данный интерфейс используется для связывания во время выполнения. Если же нет, тогда связывание проходит в основном так же, как в ситуации, когда компилятору известен тип динамического объекта времени выполнения. Две упомянутые альтернативы называются *специальным связыванием* и *языковым связыванием*.

Взаимодействие с COM можно считать третьим видом динамического связывания (см. главу 25).

## Специальное связывание

Специальное связывание происходит, когда динамический объект реализует интерфейс IDynamicMetaObjectProvider (IDMOP). Хотя интерфейс IDMOP можно реализовать в типах, которые вы пишете на языке C#, и поступать так удобно, более распространенный случай предусматривает запрос объекта, реализующего IDMOP, из динамического языка, который внедряется в .NET через исполняющую среду динамического языка (Dynamic Language Runtime – DLR), скажем, IronPython или IronRuby. Объекты из таких языков неявно реализуют интерфейс IDMOP в качестве способа для прямого управления смыслом выполняемых над ними операций.

Мы детально обсудим специальные средства привязки в главе 20, но в целях демонстрации сейчас рассмотрим простое средство привязки:

```
using System;
using System.Dynamic;

public class Test
{
  static void Main()
  {
    dynamic d = new Duck();
    d.Quack();   // Вызван метод Quack
    d.Waddle();  // Вызван метод Waddle
  }
}

public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

Класс Duck на самом деле не имеет метода Quack. Взамен он применяет специальное связывание для перехвата и интерпретации всех обращений к методам.

## Языковое связывание

Языковое связывание происходит, если динамический объект не реализует интерфейс IDMOP. Языковое связывание удобно, когда приходится иметь дело с неудачно спроектированными типами или врожденными ограничениями системы типов .NET (в главе 20 будут представлены и другие сценарии). Обычной проблемой, возникающей во время работы с числовыми типами, является отсутствие общего интерфейса. Ранее было показано, что методы могут быть привязаны динамически; то же самое справедливо и для операций:

```
static dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;
static void Main()
{
  int x = 3, y = 4;
  Console.WriteLine (Mean (x, y));
}
```

Преимущество очевидно – не приходится дублировать код для каждого числового типа. Тем не менее, утрачивается безопасность типов, повышая риск генерации исключений во время выполнения вместо получения ошибок на этапе компиляции.

Динамическое связывание обходит статическую безопасность типов, но не динамическую безопасность типов. В отличие от рефлексии (глава 19) динамическое связывание не позволяет обойти правила доступности членов.

Согласно проектному решению языковое связывание во время выполнения ведет себя максимально похоже на статическое связывание, как будто типы времени выполнения динамических объектов были известны еще на этапе компиляции. Если в предыдущем примере жестко закодировать метод Mean для работы с типом int, то поведение программы останется идентичным. Наиболее заметным исключением при проведении аналогии между статическим и динамическим связыванием являются расширяющие методы, которые мы рассмотрим в разделе "Невызываемые функции" далее в главе.

Динамическое связывание также приводит к снижению производительности. Однако из-за механизмов кеширования среды DLR повторяющиеся обращения к одному и тому же динамическому выражению оптимизируются, позволяя эффективно работать с динамическими выражениями в цикле. Такая оптимизация доводит типичные накладные расходы при выполнении простого динамического выражения на современном оборудовании до менее 100 наносекунд.

## Исключение RuntimeBinderException

Если привязка к члену не удается, тогда генерируется исключение RuntimeBinderException. Его можно считать ошибкой этапа компиляции, перенесенной на время выполнения:

```
dynamic d = 5;
d.Hello();  // Генерируется исключение RuntimeBinderException
```

Исключение генерируется из-за того, что тип int не имеет метода Hello.

## Представление типа dynamic во время выполнения

Между типами dynamic и object имеется глубокая эквивалентность. Исполняющая среда трактует следующее выражение как true:

```
typeof (dynamic) == typeof (object)
```

Данный принцип распространяется на составные типы и массивы:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Подобно объектной ссылке динамическая ссылка может указывать на объект любого типа (кроме типов указателей):

```
dynamic x = "hello";
Console.WriteLine (x.GetType().Name);  // String
x = 123;  // Ошибки нет (несмотря на то, что переменная та же самая)
Console.WriteLine (x.GetType().Name);  // Int32
```

Структурно какие-либо отличия между объектной ссылкой и динамической ссылкой отсутствуют. Динамическая ссылка просто разрешает выполнение динамических операций над объектом, на который она указывает. Чтобы выполнить любую динамическую операцию над object, тип object можно преобразовать в dynamic:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("hello");
Console.WriteLine (o);  // hello
```

При выполнении рефлексии для типа, предлагающего (открытые) члены dynamic, обнаруживается, что такие члены представлены как аннотированные члены типа object. Например, следующее определение:

```
public class Test
{
  public dynamic Foo;
}
```

эквивалентно такому:

```
public class Test
{
  [System.Runtime.CompilerServices.DynamicAttribute]
  public object Foo;
}
```

Это позволяет потребителям типа знать, что член Foo должен трактоваться как dynamic, а другим языкам, не поддерживающим динамическое связывание, работать с ним как с object.

## Динамические преобразования

Тип dynamic поддерживает неявные преобразования в и из всех остальных типов:

```
int i = 7;
dynamic d = i;
long j = d;  // Приведение не требуется (неявное преобразование)
```

Чтобы преобразование прошло успешно, тип времени выполнения динамического объекта должен быть неявно преобразуемым в целевой статический тип. Предшествующий пример работает по той причине, что тип `int` неявно преобразуем в `long`.

В следующем примере генерируется исключение `RuntimeBinderException`, т.к. тип `int` не может быть неявно преобразован в `short`:

```
int i = 7;
dynamic d = i;
short j = d;  // Генерируется исключение RuntimeBinderException
```

## Сравнение `var` и `dynamic`

Несмотря на внешнее сходство типов `var` и `dynamic`, разница между ними существенна:

- `var` говорит: позволить *компилятору* выяснить тип;
- `dynamic` говорит: позволить *исполняющей среде* выяснить тип.

Ниже показана иллюстрация:

```
dynamic x = "hello"; // Статическим типом является dynamic,
                     // а типом времени выполнения - string
var y = "hello";     // Статическим типом является string,
                     // а типом времени выполнения - string
int i = x; // Ошибка во время выполнения (невозможно преобразовать string в int)
int j = y; // Ошибка на этапе компиляции (невозможно преобразовать string в int)
```

Статическим типом переменной, объявленной с ключевым словом `var`, может быть `dynamic`:

```
dynamic x = "hello";
var y = x; // Статическим типом y является dynamic
int z = y; // Ошибка во время выполнения (невозможно преобразовать string в int)
```

## Динамические выражения

Поля, свойства, методы, события, конструкторы, индексаторы, операции и преобразования могут вызываться динамически.

Попытка потребления результата динамического выражения с возвращаемым типом `void` пресекается — точно как в случае статически типизированного выражения. Отличие связано с тем, что ошибка возникает во время выполнения:

```
dynamic list = new List<int>();
var result = list.Add (5); // Генерируется исключение RuntimeBinderException
```

Выражения, содержащие динамические операнды, обычно сами являются динамическими, т.к. эффект отсутствия информации о типе имеет каскадный характер:

```
dynamic x = 2;
var y = x * 3;  // Статическим типом y является dynamic
```

Из такого правила существует пара очевидных исключений. Во-первых, приведение динамического выражения к статическому типу дает статическое выражение:

```
dynamic x = 2;
var y = (int)x;  // Статическим типом y является int
```

Во-вторых, вызовы конструкторов всегда дают статические выражения – даже если они производятся с динамическими аргументами. В следующем примере переменная x статически типизирована как StringBuilder:

```
dynamic capacity = 10;
var x = new System.Text.StringBuilder (capacity);
```

Кроме того, существует несколько краевых случаев, когда выражение, содержащее динамический аргумент, является статическим, включая передачу индекса массиву и выражения для создания делегатов.

## Динамические вызовы без динамических получателей

В каноническом сценарии использования dynamic участвует динамический *получатель*. Это значит, что получателем динамического вызова функции является динамический объект:

```
dynamic x = ...;
x.Foo();  // x является получателем
```

Тем не менее, статически известные функции можно также вызывать с динамическими аргументами. Такие вызовы распознаются динамической перегрузкой и могут включать:

- статические методы;
- конструкторы экземпляра;
- методы экземпляра на получателях со статически известным типом.

В приведенном ниже примере конкретный метод Foo, который привязывается динамически, зависит от типа времени выполнения динамического аргумента:

```
class Program
{
  static void Foo (int x)    { Console.WriteLine ("1"); }
  static void Foo (string x) { Console.WriteLine ("2"); }

  static void Main()
  {
    dynamic x = 5;
    dynamic y = "watermelon";

    Foo (x);  // 1
    Foo (y);  // 2
  }
}
```

Поскольку динамический получатель не задействован, компилятор может статически выполнить базовую проверку успешности динамического вызова. Он проверяет, существует ли функция с правильным именем и корректным количеством параметров. Если кандидаты не найдены, тогда возникает ошибка на этапе компиляции. Например:

```
class Program
{
  static void Foo (int x)    { Console.WriteLine ("1"); }
  static void Foo (string x) { Console.WriteLine ("2"); }

  static void Main()
  {
```

```
    dynamic x = 5;
    Foo (x, x);  // Ошибка компиляции - неправильное количество параметров
    Fook (x);    // Ошибка компиляции - метод с таким именем отсутствует
  }
}
```

## Статические типы в динамических выражениях

Тот факт, что динамические типы применяются в динамическом связывании, вполне очевиден. Однако участие в динамическом связывании также и статических типов не настолько очевидно. Взгляните на следующий код:

```
class Program
{
  static void Foo (object x, object y) { Console.WriteLine ("oo"); }
  static void Foo (object x, string y) { Console.WriteLine ("os"); }
  static void Foo (string x, object y) { Console.WriteLine ("so"); }
  static void Foo (string x, string y) { Console.WriteLine ("ss"); }

  static void Main()
  {
    object o = "hello";
    dynamic d = "goodbye";
    Foo (o, d);  // os
  }
}
```

Вызов `Foo(o,d)` привязывается динамически, т.к. один из его аргументов, `d`, определен как `dynamic`. Но поскольку переменная `o` статически известна, связывание — хотя оно происходит динамически — будет использовать именно ее. В данном случае механизм распознавания перегруженных версий выберет вторую реализацию `Foo` из-за статического типа `o` и типа времени выполнения `d`. Другими словами, компилятор является "настолько статическим, насколько это возможно".

## Невызываемые функции

Некоторые функции не могут быть вызваны динамически. Нельзя вызывать динамически:

- расширяющие методы (через синтаксис расширяющих методов);
- члены интерфейса, если для этого необходимо выполнить приведение к данному интерфейсу;
- члены базового класса, которые скрыты подклассом.

Понимание причин важно для понимания динамического связывания в целом.

Динамическое связывание требует двух порций информации: имени вызываемой функции и объекта, на котором должна вызываться функция. Тем не менее, в каждом из трех невызываемых сценариев присутствует *дополнительный тип*, который известен только на этапе компиляции. Начиная с версии C# 6, какие-то способы динамически указывать такие дополнительные типы отсутствуют.

При вызове расширяющих методов этот дополнительный тип является неявным. Он представляет собой статический класс, в котором определен расширяющий метод. Компилятор ищет его с учетом директив `using`, присутствующих в исходном коде. В результате расширяющие методы превращаются в концепции, существующие только на этапе компиляции, т.к. после компиляции директивы `using` исчезают

(после завершения своей работы в рамках процесса связывания, которая заключается в отображении простых имен на имена, уточненные пространствами имен).

При вызове членов через интерфейс дополнительный тип указывается через неявное или явное приведение. Существуют два сценария, когда подобное может пригодиться: при вызове явно реализованных членов интерфейса и при вызове членов интерфейса, реализованных в типе, который является внутренним в другой сборке. Второй сценарий можно проиллюстрировать с помощью следующих двух типов:

```
interface IFoo   { void Test();        }
class Foo : IFoo { void IFoo.Test() {} }
```

Для вызова метода `Test` потребуется выполнить приведение к интерфейсу `IFoo`. При статической типизации это делается легко:

```
IFoo f = new Foo();  // Неявное приведение к типу интерфейса
f.Test();
```

А теперь рассмотрим ситуацию с динамической типизацией:

```
IFoo f = new Foo();
dynamic d = f;
d.Test();            // Генерируется исключение
```

Выделенное полужирным неявное приведение сообщает *компилятору* о необходимости привязки последующих вызовов членов `f` к интерфейсу `IFoo`, а не к `Foo` – другими словами, для просмотра данного объекта сквозь призму `IFoo`. Однако во время выполнения такая призма теряется, а потому среда DLR не может завершить связывание. Упомянутая потеря продемонстрирована ниже:

```
Console.WriteLine (f.GetType().Name);  // Foo
```

Похожая ситуация возникает при вызове сокрытого члена базового класса: дополнительный тип должен указываться либо через приведение, либо с помощью ключевого слова `base` – и этот дополнительный тип утрачивается во время выполнения.

# Перегрузка операций

Операции могут быть перегружены для предоставления специальным типам более естественного синтаксиса. Перегрузку операций целесообразнее всего применять при реализации специальных структур, которые представляют относительно примитивные типы данных. Например, хорошим кандидатом на перегрузку операций может служить специальный числовой тип.

Разрешено перегружать следующие символические операции:

| | | | | |
|---|---|---|---|---|
| + (унарная) | - (унарная) | ! | ~ | ++ |
| -- | + | - | * | / |
| % | & | \| | ^ | << |
| >> | == | != | > | < |
| >= | <= | | | |

Перечисленные ниже операции также могут быть перегружены:

- явные и неявные преобразования (с использованием ключевых слов `explicit` и `implicit`);
- *операции* `true` и `false` (не *литералы*).

Следующие операции перегружаются косвенно:

- составные операции присваивания (например, +=, /=) неявно перегружаются при перегрузке обычных операций (т.е. +, /);
- условные операции && и || неявно перегружаются при перегрузке побитовых операций & и |.

## Функции операций

Операция перегружается за счет объявления *функции операции* (operator). Функция операции подчиняется перечисленным далее правилам.

- Имя функции указывается с помощью ключевого слова operator, за которым следует символ операции.
- Функция операции должна быть помечена как static и public.
- Параметры функции операции представляют операнды.
- Возвращаемый тип функции операции представляет результат выражения.
- По меньшей мере, один из операндов должен иметь тип, для которого объявляется функция операции.

В следующем примере мы определяем структуру по имени Note, представляющую музыкальную ноту, и затем перегружаем операцию +:

```
public struct Note
{
  int value;
  public Note (int semitonesFromA) { value = semitonesFromA; }
  public static Note operator + (Note x, int semitones)
  {
    return new Note (x.value + semitones);
  }
}
```

Перегруженная версия операции + позволяет добавлять значение int к Note:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Перегрузка операции приводит к автоматической перегрузке соответствующей составной операции присваивания. Поскольку в примере перегружена операция +, можно также применять операцию +=:

```
CSharp += 2;
```

Начиная с версии C# 6, подобно методам и свойства функции операций, состоящие из одиночного выражения, разрешено записывать более компактно с помощью синтаксиса функций, сжатых до выражений:

```
public static Note operator + (Note x, int semitones)
                          => new Note (x.value + semitones);
```

## Перегрузка операций эквивалентности и сравнения

Операции эквивалентности и сравнения часто перегружаются при написании структур и в редких случаях — при написании классов. При перегрузке операций эквивалентности и сравнения должны соблюдаться специальные правила и обязательства, которые будут подробно рассматриваться в главе 6.

Ниже приведен краткий обзор этих правил.

*Парность*

Компилятор C# требует, чтобы операции, которые представляют собой логические пары, были определены обе. Такими операциями являются (== !=), (< >) и (<= >=).

`Equals` и `GetHashCode`

В большинстве случаев при перегрузке операций == и != обычно необходимо переопределять методы `Equals` и `GetHashCode` класса `object`, чтобы обеспечить осмысленное поведение. Компилятор C# выдаст предупреждение, если это не сделано. (За дополнительными сведениями обращайтесь в раздел "Сравнение эквивалентности" главы 6.)

`IComparable` и `IComparable<T>`

Если перегружаются операции (< >) и (<= >=), тогда обязательно должны быть реализованы интерфейсы `IComparable` и `IComparable<T>`.

## Специальные неявные и явные преобразования

Неявные и явные преобразования являются перегружаемыми операциями. Как правило, они перегружаются для того, чтобы сделать преобразования между тесно связанными типами (такими как числовые типы) лаконичными и естественными.

Для преобразования между слабо связанными типами больше подходят следующие стратегии:

- написать конструктор, принимающий параметр типа, из которого выполняется преобразование;
- написать методы `To`*XXX* и (статические) методы `From`*XXX*, предназначенные для преобразования между типами.

Как объяснялось при обсуждении типов, логическое обоснование неявных преобразований заключается в том, что они гарантированно выполняются успешно и не приводят к потере информации. И наоборот, явное преобразование должно быть обязательным либо когда успешность преобразования определяется обстоятельствами во время выполнения, либо если в результате преобразования может быть потеряна информация.

В показанном далее примере мы определяем преобразования между типом `Note` и типом `double` (с использованием которого представлена частота в герцах данной ноты):

```
...
// Преобразование в герцы
public static implicit operator double (Note x)
  => 440 * Math.Pow (2, (double) x.value / 12 );

// Преобразование из герц (с точностью до ближайшего полутона)
public static explicit operator Note (double x)
  => new Note ((int) (0.5 + 12 * (Math.Log (x/440) / Math.Log(2) ) ));
...
Note n = (Note)554.37;    // явное преобразование
double x = n;             // неявное преобразование
```

Следуя нашим собственным принципам, этот пример можно реализовать более эффективно с помощью метода `ToFrequency` (и статического метода `FromFrequency`) вместо неявной и явной операции.

Операции `as` и `is` игнорируют специальные преобразования:

```
Console.WriteLine (554.37 is Note);      // False
Note n = 554.37 as Note;                 // Ошибка
```

## Перегрузка операций `true` и `false`

Операции `true` и `false` перегружаются в исключительно редких случаях для типов, которые являются булевскими "по духу", но не имеют преобразования в `bool`. Примером может служить тип, реализующий логику с тремя состояниями: за счет перегрузки `true` и `false` этот тип может гладко работать с условными операторами и операциями, а именно — `if`, `do`, `while`, `for`, `&&`, `||` и `?:`. Такую функциональность предоставляет структура `System.Data.SqlTypes.SqlBoolean`. Например:

```
SqlBoolean a = SqlBoolean.Null;
if (a)
  Console.WriteLine ("True");
else if (!a)
  Console.WriteLine ("False");
else
  Console.WriteLine ("Null");
```

```
ВЫВОД:
Null
```

Приведенный ниже код представляет собой повторную реализацию частей структуры `SqlBoolean`, необходимой для демонстрации работы с операциями `true` и `false`:

```
public struct SqlBoolean
{
  public static bool operator true (SqlBoolean x)
    => x.m_value == True.m_value;

  public static bool operator false (SqlBoolean x)
    => x.m_value == False.m_value;

  public static SqlBoolean operator ! (SqlBoolean x)
  {
    if (x.m_value == Null.m_value)  return Null;
    if (x.m_value == False.m_value) return True;
    return False;
  }

  public static readonly SqlBoolean Null =  new SqlBoolean(0);
  public static readonly SqlBoolean False = new SqlBoolean(1);
  public static readonly SqlBoolean True =  new SqlBoolean(2);

  private SqlBoolean (byte value) { m_value = value; }
  private byte m_value;
}
```

# Небезопасный код и указатели

Язык C# поддерживает прямые манипуляции с памятью через указатели внутри блоков кода, которые помечены как небезопасные и скомпилированы с опцией компилятора `/unsafe`. Типы указателей полезны главным образом при взаимодействии с API-интерфейсами C, но могут также применяться для доступа в память за пределами управляемой кучи или для "горячих" точек, критичных к производительности.

## Основы указателей

Для каждого типа значения или ссылочного типа `V` имеется соответствующий тип указателя `V*`. Экземпляр указателя хранит адрес переменной. Тип указателя может быть (небезопасно) приведен к любому другому типу указателя. Ниже описаны основные операции над указателями.

| Операция | Описание |
|---|---|
| `&` | Операция *взятия адреса* возвращает указатель на адрес переменной |
| `*` | Операция *разыменования* возвращает переменную, находящуюся по адресу, заданному указателем |
| `->` | Операция *указателя на член* является синтаксическим сокращением, т.е. `x->y` эквивалентно `(*x).y` |

## Небезопасный код

Помечая тип, член типа или блок операторов ключевым словом `unsafe`, вы разрешаете внутри этой области видимости использовать типы указателей и выполнять операции над указателями в стиле C++. Ниже показан пример применения указателей для быстрой обработки битовой карты:

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

Небезопасный код может выполняться быстрее, чем соответствующая ему безопасная реализация. В последнем случае код потребует вложенного цикла с индексацией в массиве и проверкой границ. Небезопасный метод C# может также оказаться быстрее, чем вызов внешней функции C, поскольку не будет никаких накладных расходов, связанных с покиданием управляемой среды выполнения.

## Оператор `fixed`

Оператор `fixed` необходим для закрепления управляемого объекта, такого как битовая карта в предыдущем примере. Во время выполнения программы многие объекты распределяются в куче и впоследствии освобождаются. Во избежание нежелательных затрат или фрагментации памяти сборщик мусора перемещает объекты внутри кучи. Указатель на объект бесполезен, если адрес объекта может измениться, пока на него производится ссылка, потому оператор `fixed` сообщает сборщику мусора о необходимости "закрепления" объекта, чтобы он никуда не перемещался. Это может оказать влияние на эффективность программы во время выполнения, так что фиксированные блоки должны использоваться только кратковременно, а внутри фиксированного блока следует избегать распределения памяти в куче.

В рамках оператора `fixed` можно получать указатель на любой тип значения, массив типов значений или строку. В случае массивов и строк указатель будет в действительности указывать на первый элемент, который относится к типу значения.

Типы значений, объявленные внутри ссылочных типов, требуют закрепления ссылочных типов, как показано ниже:

```
class Test
{
  int x;
  static void Main()
  {
    Test test = new Test();
    unsafe
    {
      fixed (int* p = &test.x)  // Закрепляет test
      {
        *p = 9;
      }
      System.Console.WriteLine (test.x);
    }
  }
}
```

Более подробно оператор `fixed` рассматривается в разделе "Отображение структуры на неуправляемую память" главы 25.

## Операция указателя на член

В дополнение к операциям `&` и `*` язык C# также предлагает операцию `->` в стиле C++, которая может применяться при работе со структурами:

```
struct Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    Test* p = &test;
    p->x = 9;
    System.Console.WriteLine (test.x);
  }
}
```

## Массивы

### Ключевое слово stackalloc

Память может быть выделена в блоке внутри стека явно с использованием ключевого слова `stackalloc`. Из-за распределения в стеке время жизни блока памяти ограничивается выполнением метода, в точности как для любой другой локальной переменной. К данному блоку можно применять операцию `[]` для проведения индексации в рамках памяти:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
  Console.WriteLine (a[i]); // Вывод на экран низкоуровневых значений из памяти
```

### Буферы фиксированных размеров

С помощью ключевого слова `fixed` можно также выделять память в блоке внутри структур:

```
unsafe struct UnsafeUnicodeString
{
  public short Length;
  public fixed byte Buffer[30];  // Выделить блок из 30 байтов
}

unsafe class UnsafeClass
{
  UnsafeUnicodeString uus;

  public UnsafeClass (string s)
  {
    uus.Length = (short)s.Length;
    fixed (byte* p = uus.Buffer)
      for (int i = 0; i < s.Length; i++)
        p[i] = (byte) s[i];
  }
}
class Test
{
  static void Main() { new UnsafeClass ("Christian Troy"); }
}
```

В этом примере ключевое слово `fixed` используется еще и для закрепления в куче объекта, содержащего буфер (который будет экземпляром `UnsafeClass`). Таким образом, ключевое слово `fixed` имеет два разных смысла: фиксация *размера* и фиксация *места*. Оба случая применения часто встречаются вместе, поскольку для использования буфера фиксированного размера он должен быть зафиксирован на месте.

## void*

*Указатель void* (`void*`) не делает никаких предположений о типе лежащих в основе данных и удобен для функций, которые имеют дело с низкоуровневой памятью. Существует неявное преобразование из любого типа указателя в `void*`. Указатель `void*` не допускает разыменования и выполнения над ним арифметических операций. Например:

```
class Test
{
```

```
  unsafe static void Main()
  {
    short[ ] a = {1,1,2,3,5,8,13,21,34,55};
    fixed (short* p = a)
    {
      // Операция sizeof возвращает размер типа значения в байтах
      Zap (p, a.Length * sizeof (short));
    }
    foreach (short x in a)
      System.Console.WriteLine (x);  // Выводит все нули
  }
  unsafe static void Zap (void* memory, int byteCount)
  {
    byte* b = (byte*) memory;
      for (int i = 0; i < byteCount; i++)
        *b++ = 0;
  }
}
```

## Указатели на неуправляемый код

Указатели также удобны для доступа к данным, находящимся за пределами управляемой кучи (например, при взаимодействии с DLL-библиотеками C или COM), либо при работе с данными, которые хранятся не в основной памяти (скажем, в памяти графического адаптера или на носителе встроенного устройства).

# Директивы препроцессора

Директивы препроцессора снабжают компилятор дополнительной информацией о разделах кода. Наиболее распространенными директивами препроцессора являются директивы условной компиляции, которые предоставляют способ включения либо исключения разделов кода из процесса компиляции. Например:

```
#define DEBUG
class MyClass
{
  int x;
  void Foo()
  {
    #if DEBUG
    Console.WriteLine ("Testing: x = {0}", x);
    #endif
  }
  ...
}
```

В классе `MyClass` оператор внутри метода `Foo` компилируется условно в зависимости от существования символа `DEBUG`. Если удалить определение символа `DEBUG`, тогда этот оператор в `Foo` компилироваться не будет. Символы препроцессора могут определяться внутри файла исходного кода (что и было сделано в примере), а также передаваться компилятору с помощью опции командной строки `/define:`*символ*.

В директивах `#if` и `#elif` можно применять операции `||`, `&&` и `!` для выполнения логических действий *ИЛИ*, *И* и *НЕ* над несколькими символами.

Представленная ниже директива указывает компилятору на необходимость включения следующего за ней кода, если определен символ TESTMODE и не определен символ DEBUG:

```
#if TESTMODE && !DEBUG
  ...
```

Тем не менее, имейте в виду, что вы не строите обычное выражение C#, а символы, которыми вы оперируете, не имеют абсолютно никакого отношения к *переменным* — статическим или каким-то другим.

Директивы `#error` и `#warning` предотвращают случайное неправильное использование директив условной компиляции, заставляя компилятор генерировать предупреждение или сообщение об ошибке, которое вызвано неподходящим набором символов компиляции. Директивы препроцессора перечислены в табл. 4.1.

**Таблица 4.1. Директивы препроцессора**

| Директива препроцессора | Действие |
|---|---|
| `#define` *символ* | Определяет символ |
| `#undef` *символ* | Отменяет определение символа |
| `#if` *символ* [*операция символ2*]... | Условная компиляция; операциями являются `==`, `!=`, `&&` и `\|\|`, за которыми следуют директивы `#else`, `#elif` и `#endif` |
| `#else` | Компилирует код до следующей директивы `#endif` |
| `#elif` *символ* [*операция символ2*] | Комбинирует ветвь `#else` и проверку `#if` |
| `#endif` | Заканчивает директивы условной компиляции |
| `#warning` *текст* | Заставляет компилятор вывести предупреждение с указанным текстом |
| `#error` *текст* | Заставляет компилятор вывести сообщение об ошибке с указанным текстом |
| `#pragma warning [disable \| restore]` | Отключает или восстанавливает выдачу компилятором предупреждения (предупреждений) |
| `#line [`*номер* `["`*файл*`"] \| hidden]` | Номер задает строку в исходном коде; в "*файл*" указывается имя файла для помещения в вывод компилятора; `hidden` инструктирует инструменты отладки о необходимости пропуска кода от этой точки до следующей директивы `#line` |
| `#region` *имя* | Обозначает начало раздела |
| `#endregion` | Обозначает конец раздела |

## Условные атрибуты

Атрибут, декорированный атрибутом `Conditional`, будет компилироваться, только если определен заданный символ препроцессора. Например:

```
// file1.cs
#define DEBUG
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
```

```
public class TestAttribute : Attribute {}

// file2.cs
#define DEBUG
[Test]
class Foo
{
  [Test]
  string s;
}
```

Компилятор включит атрибуты `[Test]`, только если символ `DEBUG` определен в области видимости для файла `file2.cs`.

## Директива `#pragma warning`

Компилятор генерирует предупреждение, когда обнаруживает в коде что-то, выглядящее непреднамеренным. В отличие от ошибок предупреждения обычно не препятствуют компиляции приложения.

Предупреждения компилятора могут быть исключительно полезными при выявлении ошибок. Тем не менее, их полезность снижается в случае выдачи *ложных* предупреждений. Чтобы заметить "настоящие" предупреждения в крупном приложении, важно поддерживать подходящее соотношение "сигнал-шум".

С этой целью компилятор позволяет избирательно подавлять выдачу предупреждений посредством директивы `#pragma warning`. В следующем примере мы инструктируем компилятор не выдавать предупреждения о том, что поле `Message` не применяется:

```
public class Foo
{
  static void Main() { }

  #pragma warning disable 414
  static string Message = "Hello";
  #pragma warning restore 414
}
```

Отсутствие числа в директиве `#pragma warning` означает, что будет отключена или восстановлена выдача предупреждений со всеми кодами.

Если вас интересует всестороннее использование данной директивы, тогда можете скомпилировать код с переключателем `/warnaserror`, который сообщит компилятору о необходимости трактовать любые оставшиеся предупреждения как ошибки.

# XML-документация

*Документирующий комментарий* – это порция встроенного XML-кода, которая документирует тип или член типа. Документирующий комментарий располагается непосредственно перед объявлением типа или члена и начинается с трех символов косой черты:

```
/// <summary>Прекращает выполняющийся запрос.</summary>
public void Cancel() { ... }
```

Многострочные комментарии записываются следующим образом:

```
/// <summary>
/// Прекращает выполняющийся запрос.
/// </summary>
public void Cancel() { ... }
```

или так (обратите внимание на дополнительный символ звездочки в начале):

```
/**
   <summary>Прекращает выполняющийся запрос.</summary>
*/
public void Cancel() { ... }
```

В случае компиляции с переключателем /doc (в среде Visual Studio перейдите на вкладку Build (Сборка) диалогового окна Project Properties (Свойства проекта)) компилятор извлекает и накапливает документирующие комментарии в специальном XML-файле. С данным файлом связаны два основных сценария работы.

- Если он размещен в той же папке, что и скомпилированная сборка, то Visual Studio автоматически читает этот XML-файл и применяет информацию из него для предоставления списка членов IntelliSense потребителям сборки с таким же именем, как у XML-файла.
- Сторонние инструменты (такие как Sandcastle и NDoc) могут трансформировать этот XML-файл в справочный HTML-файл.

## Стандартные XML-дескрипторы документации

Ниже перечислены стандартные XML-дескрипторы, которые распознаются Visual Studio и генераторами документации.

**<summary>**

```
<summary>...</summary>
```

Указывает всплывающую подсказку, которую средство IntelliSense должно отображать для типа или члена; обычно это одиночное выражение или предложение.

**<remarks>**

```
<remarks>...</remarks>
```

Дополнительный текст, который описывает тип или член. Генераторы документации объединяют его с полным описанием типа или члена.

**<param>**

```
<param name="имя">...</param>
```

Объясняет параметр метода.

**<returns>**

```
<returns>...</returns>
```

Объясняет возвращаемое значение метода.

**<exception>**

```
<exception [cref="тип"]>...</exception>
```

Указывает исключение, которое метод может генерировать (в cref задается тип исключения).

**<permission>**

```
<permission [cref="тип"]>...</permission>
```

Указывает тип IPermission, требуемый документируемым типом или членом.

**<example>**

```
<example>...</example>
```

Обозначает пример (используемый генераторами документации). Как правило, содержит описательный текст и исходный код (исходный код обычно заключен в дескриптор <c> или <code>).

**<c>**

```
<c>...</c>
```

Указывает внутристрочный фрагмент кода. Этот дескриптор обычно применяется внутри блока <example>.

**<code>**

```
<code>...</code>
```

Указывает многострочный пример кода. Этот дескриптор обычно используется внутри блока <example>.

**<see>**

```
<see cref="член">...</see>
```

Вставляет внутристрочную перекрестную ссылку на другой тип или член. Генераторы HTML-документации обычно преобразуют ее в гиперссылку. Компилятор выдает предупреждение, если указано недопустимое имя типа или члена.

**<seealso>**

```
<seealso cref="член">...</seealso>
```

Вставляет перекрестную ссылку на другой тип или член. Генераторы документации обычно помещают ее внутрь отдельного раздела "See Also" ("См. также") в нижней части страницы.

**<paramref>**

```
<paramref name="имя"/>
```

Вставляет ссылку на параметр внутри дескриптора <summary> или <remarks>.

**<list>**

```
<list type=[ bullet | number | table ]>
  <listheader>
    <term>...</term>
    <description>...</description>
  </listheader>
  <item>
    <term>...</term>
    <description>...</description>
  </item>
</list>
```

Инструктирует генератор документации о необходимости выдачи маркированного (bullet), нумерованного (number) или табличного (table) списка.

**<para>**

```
<para>...</para>
```

Инструктирует генератор документации о необходимости форматирования содержимого в виде отдельного абзаца.

**<include>**

```
<include file='имя-файла' path='путь-к-дескриптору[@name="идентификатор"]'>
  ...
</include>
```

Выполняет объединение с внешним XML-файлом, содержащим документацию. В атрибуте `path` задается XPath-запрос к конкретному элементу из этого файла.

## Дескрипторы, определяемые пользователем

С предопределенными XML-дескрипторами, распознаваемыми компилятором C#, не связано ничего особенного, и вы можете также определять собственные дескрипторы. Компилятор предпринимает специальную обработку только для дескриптора `<param>` (проверяет имя параметра, а также выясняет, документированы ли все параметры метода) и атрибута `cref` (проверяет, что атрибут ссылается на реальный тип или член, и расширяет его в полностью заданный идентификатор типа или члена). Атрибут `cref` можно также применять в собственных дескрипторах – он будет проверен и расширен в точности, как для предопределенных дескрипторов `<exception>`, `<permission>`, `<see>` и `<seealso>`.

## Перекрестные ссылки на типы или члены

Имена типов и перекрестные ссылки на типы или члены транслируются в идентификаторы, уникальным образом определяющие тип или член. Такие имена образованы из префикса, который определяет, что конкретно представляет идентификатор, и сигнатуры типа или члена. Префиксы членов перечислены ниже.

| XML-префикс типа | К какому идентификатору применяется |
|---|---|
| N | Пространство имен |
| T | Тип (класс, структура, перечисление, интерфейс, делегат) |
| F | Поле |
| P | Свойство (включая индексаторы) |
| M | Метод (включая специальные методы) |
| E | Событие |
| ! | Ошибка |

Правила генерации сигнатур хорошо документированы, хотя и довольно сложны. Вот пример типа и сгенерированных идентификаторов:

```
// Пространства имен не имеют независимых сигнатур
namespace NS
{
  /// T:NS.MyClass
  class MyClass
  {
    /// F:NS.MyClass.aField
    string aField;

    /// P:NS.MyClass.aProperty
    short aProperty {get {...} set {...}}
```

```
    /// T:NS.MyClass.NestedType
    class NestedType {...};

    /// M:NS.MyClass.X()
    void X() {...}

    /// M:NS.MyClass.Y(System.Int32,System.Double@,System.Decimal@)
    void Y(int p1, ref double p2, out decimal p3) {...}

    /// M:NS.MyClass.Z(System.Char[ ],System.Single[0:,0:])
    void Z(char[ ] 1, float[,] p2) {...}

    /// M:NS.MyClass.op_Addition(NS.MyClass,NS.MyClass)
    public static MyClass operator+(MyClass c1, MyClass c2) {...}

    /// M:NS.MyClass.op_Implicit(NS.MyClass)~System.Int32
    public static implicit operator int(MyClass c) {...}

    /// M:NS.MyClass.#ctor
    MyClass() {...}

    /// M:NS.MyClass.Finalize
    ~MyClass() {...}

    /// M:NS.MyClass.#cctor
    static MyClass() {...}
  }
}
```

# 5

# Обзор .NET Framework

Почти все возможности .NET Framework доступны через обширное множество управляемых типов. Эти типы организованы в иерархические пространства имен и упакованы в набор сборок, которые вместе со средой CLR образуют платформу .NET.

Некоторые типы .NET используются напрямую CLR и являются критически важными для среды управляемого размещения. Такие типы находятся в сборке по имени *mscorlib.dll* и включают встроенные типы C#, а также базовые классы коллекций, типы для обработки потоков данных, сериализации, рефлексии, многопоточности и собственной возможности взаимодействия ("mscorlib" представляет собой аббревиатуру от Multi-language Standard Common Object Runtime Library – стандартная многоязыковая общая объектная библиотека времени выполнения).

Уровнем выше находятся дополнительные типы, которые расширяют функциональность уровня CLR, предоставляя такие средства, как XML, работа в сети и LINQ. Они находятся в сборках *System.dll*, *System.Xml.dll* и *System.Core.dll*, и совместно с mscorlib.dll формируют развитую среду для программирования, на основе которой построены остальные части .NET Framework. Такая "центральная инфраструктура" в значительной степени определяет контекст оставшихся глав настоящей книги.

Остаток инфраструктуры .NET Framework состоит из прикладных API-интерфейсов, большинство из которых охватывают три области функциональности:

- технологии пользовательских интерфейсов;
- технологии серверной части;
- технологии распределенных систем.

В табл. 5.1 показана хронология совместимости между версиями C#, CLR и .NET Framework.

В главе приведен обзор всех ключевых областей .NET Framework, начиная с раскрытых в настоящей книге основных типов и заканчивая краткими сведениями о прикладных технологиях.

**Таблица 5.1. Версии C#, CLR и .NET Framework**

| Версия C# | Версия CLR | Версии .NET Framework |
|---|---|---|
| 1.0 | 1.0 | 1.0 |
| 1.2 | 1.1 | 1.1 |
| 2.0 | 2.0 | 2.0, 3.0 |
| 3.0 | 2.0 (SP2) | 3.5 |
| 4.0 | 4.0 | 4.0 |
| 5.0 | 4.5 (исправленная версия CLR 4.0) | 4.5 |
| 6.0 | 4.6 (исправленная версия CLR 4.0) | 4.6 |
| 7.0 | 4.6/4.7 (исправленная версия CLR 4.0) | 4.6/4.7 |

## Нововведения версии .NET Framework 4.6

- Сборщик мусора (Garbage Collector – GC) предлагает больший контроль над тем, когда (не) производить сборку мусора, через новые методы класса `GC`. При вызове метода `GC.Collect` также доступны дополнительные параметры для более точной настройки.
- Появился совершенно новый более быстрый 64-разрядный JIT-компилятор.
- Пространство имен `System.Numerics` теперь включает аппаратно-ускоренные типы матриц, векторов, `BigInteger` и `Complex`.
- Появился новый класс `System.AppContext`, который снабжает авторов библиотек согласованным механизмом, позволяющим потребителям библиотек включать или отключать средства новых API-интерфейсов.
- Объекты `Task` при создании выбирают культуру текущего потока и культуру пользовательского интерфейса.
- Интерфейс `IReadOnlyCollection<T>` реализует большее количество типов коллекций.
- В инфраструктуру WPF внесены дополнительные усовершенствования, включая улучшенную обработку касаний и более высоких DPI.
- Инфраструктура ASP.NET теперь поддерживает HTTP/2 и протокол привязки по маркерам (Token Binding Protocol) в Windows 10.

Сборки и пространства имен в .NET Framework *пересекаются*. Наиболее экстремальными примерами считаются *`mscorlib.dll`* и *`System.Core.dll`*, где определены типы в десятках пространств имен, причем ни одно из них не начинается с *`mscorlib`* или *`System.Core`*. Однако менее очевидные случаи являются более запутанными, такие как типы в `System.Security.Cryptography`. Большинство типов в этом пространстве имен находится в сборке *`System.dll`* за исключением нескольких типов, которые расположены в сборке *`System.Security.dll`*.

На веб-сайте, посвященном книге, содержится полное отображение пространств имен .NET Framework на сборки (`http://www.albahari.com/nutshell/NamespaceReference.aspx`).

Многие ключевые типы определены в сборках `mscorlib.dll`, `System.dll` и `System.Core.dll`. Первая сборка, `mscorlib.dll`, содержит типы, которые требуются для самой исполняющей среды; в сборках `System.dll` и `System.Core.dll` находятся дополнительные типы, необходимые для программистов. Причина того, что последние две сборки являются отдельными, чисто историческая: версию .NET Framework 3.5 в Microsoft старались сделать насколько возможно *добавочной*, т.к. она запускалась в виде уровня поверх существующей среды CLR 2.0. Вследствие этого почти все новые основные типы (такие как классы, поддерживающие LINQ) вошли в новую сборку, которую в Microsoft назвали `System.Core.dll`.

### Нововведения версии .NET Framework 4.7

Версия .NET Framework 4.7 — в большей степени корректировочный, нежели вводящий новые средства выпуск, с многочисленными исправлениями дефектов и незначительными улучшениями. Кроме того, данная версия обладает следующими особенностями.

- Структура `System.ValueTuple` входит в состав .NET Framework 4.7, так что можно использовать кортежи C# 7, не ссылаясь на сборку `System.ValueTuple.dll`.
- В WPF улучшилась поддержка касаний.
- В Windows Forms улучшилась поддержка мониторов с высокими параметрами DPI.

# .NET Standard 2.0

В главе 1 мы описали три главные альтернативы .NET Framework для межплатформенной разработки:

- UWP для устройств и настольных компьютеров Windows 10;
- .NET Core/ASP.NET Core для Windows, Linux и MacOS;
- Xamarin для мобильных устройств (с iOS, Android и Windows 10).

Хорошая новость в том, что с выходом .NET Core 2.0 указанные инфраструктуры — наряду с .NET Framework 4.6.1 и последующими версиями — сблизились по своей основной функциональности и теперь все они предлагают библиотеку базовых классов (BCL) с похожими типами и членами. Такая общность была формализована как стандарт под названием *.NET Standard 2.0*.

При написании библиотеки в Visual Studio 2017 вместо специфической инфраструктуры вы можете выбрать в качестве целевой платформы *.NET Standard 2.0*. Тогда ваша библиотека станет *переносимой*, и та же самая сборка будет запускаться без модификаций под управлением (современных версий) всех четырех инфраструктур.

.NET Standard — не .NET Framework; это просто спецификация, описывающая минимальный уровень функциональности (типы и члены), который гарантирует совместимость с определенным набором инфраструктур. Концепция похожа на интерфейсы C#: стандарт .NET Standard подобен интерфейсу, который конкретные типы (инфраструктуры) могут реализовать.

В настоящей книге раскрывается большинство из того, что находится в .NET Standard 2.0.

## Старые стандарты .NET Standard

Существуют и применяются также старые стандарты .NET Standard, наиболее примечательны из которых 1.1, 1.2, 1.3 и 1.6. Стандарт с более высоким номером всегда представляет собой строгое надмножество стандарта с более низким номером. Например, при написании библиотеки, которая нацелена на .NET Standard 1.6, вы будете поддерживать не только последние версии четырех крупных инфраструктур, но также .NET Core 1.0. А если вы нацеливаете библиотеку на .NET Standard 1.3, то будет поддерживаться все, что мы уже упомянули, плюс .NET Framework 4.6.0 (табл. 5.2).

**Таблица 5.2. Старые стандарты .NET Standard**

| Если вы нацеливаете библиотеку на... | То будут также поддерживаться... |
|---|---|
| .NET Standard 1.6 | .NET Core 1.0 |
| .NET Standard 1.3 | Указанное выше плюс .NET Framework 4.6.0 |
| .NET Standard 1.2 | Указанное выше плюс .NET Framework 4.5.1, Windows Phone 8.1, WinRT для Windows 8.1 |
| .NET Standard 1.1 | Указанное выше плюс .NET Framework 4.5.0, Windows Phone 8.0, WinRT для Windows 8.0 |

В стандартах 1.x отсутствуют тысячи API-интерфейсов, которые находятся в стандарте 2.0, в том числе большинство того, что мы описываем в этой книге. В результате нацеливание на какой-то стандарт 1.x становится гораздо более затруднительным, особенно в случае необходимости интеграции с существующим кодом или библиотеками.

Если вас интересует поддержка более старых инфраструктур, но не межплатформенная совместимость, тогда лучше нацеливаться на старую версию специфической инфраструктуры. В случае Windows удачным выбором будет версия .NET Framework 4.5, т.к. она широко развернута (заранее установлена на всех машинах с Windows 8 и выше) и содержит бо́льшую часть того, что есть в .NET Framework 4.7.

Вы можете думать о стандарте .NET Standard как о наименьшем общем знаменателе. В случае стандарта .NET Standard 2.0 четыре инфраструктуры, которые ему следуют, имеют похожую библиотеку базовых классов, поэтому наименьший общий знаменатель является крупным и полезным. Тем не менее, если вам также нужна совместимость с инфраструктурой .NET Core 1.0 (с ее значительно усеченной библиотекой BCL), тогда наименьший общий знаменатель — .NET Standard 1.x — становится гораздо уже и менее полезным.

## Ссылочные сборки

При компиляции программы вы обязаны ссылаться на сборки, которые содержат части инфраструктуры, потребляемые вашей программой. Например, простая консольная программа для .NET Framework, которая включает запрос LINQ to XML, потребовала бы сборки `mscorlib.dll`, `System.dll`, `System.Xml.dll`, `System.Xml.Linq.dll` и `System.Core.dll`.

В среде Visual Studio это делается путем добавления к проекту ссылок (только что перечисленные ссылки добавляются автоматически, когда создается проект, нацеленный на .NET Framework 4.x). Однако сборки, на которые производится ссылка, должны существовать только в интересах компилятора и не обязательно будут теми же, что используются во время выполнения. Следовательно, допустимо применять специальные ссылочные сборки, которые существуют как пустые оболочки без какого-либо скомпилированного кода. Именно так работает стандарт .NET Standard: вы добавляете *ссылочную сборку* по имени `netstandard.dll`, которая содержит все допустимые типы и члены в .NET Standard 2.0 (но не фактический скомпилированный код). Затем посредством атрибутов переадресации сборок во время выполнения загружаются "реальные" сборки. (Выбор "реальных" сборок будет зависеть от того, под управлением какой инфраструктуры происходит запуск.)

Ссылочные сборки также позволяют нацеливаться на более низкую версию .NET Framework, которая установлена на вашей машине. Скажем, если вы установили .NET Framework 4.7 вместе с Visual Studio 2017, то по-прежнему можете нацеливать свой проект на .NET Framework 4.0. Благодаря набору ссылочных сборок .NET Framework 4.0 ваш проект будет способен видеть только типы/члены .NET Framework 4.0.

# Среда CLR и ядро платформы

## Системные типы

Наиболее фундаментальные типы находятся прямо в пространстве имен `System`. В их число входят встроенные типы C#, базовый класс `Exception`, базовые классы `Enum`, `Array` и `Delegate`, а также типы `Nullable`, `Type`, `DateTime`, `TimeSpan` и `Guid`. Кроме того, пространство имен `System` включает типы для выполнения математических функций (`Math`), генерации случайных чисел (`Random`) и преобразования между различными типами (`Convert` и `BitConverter`).

Фундаментальные типы описаны в главе 6 вместе с интерфейсами, которые определяют стандартные протоколы, используемые повсеместно в .NET Framework для решения таких задач, как форматирование (`IFormattable`) и сравнение порядка (`IComparable`).

В пространстве имен `System` также определен интерфейс `IDisposable` и класс `GC` для взаимодействия со сборщиком мусора. Эти темы будут раскрыты в главе 12.

## Обработка текста

Пространство имен `System.Text` содержит класс `StringBuilder` (редактируемый или *изменяемый* родственник `string`) и типы для работы с кодировками текста, такими как UTF-8 (`Encoding` и его подтипы). Мы рассмотрим их в главе 6.

Пространство имен `System.Text.RegularExpressions` содержит типы, которые выполняют расширенные операции поиска и замены на основе шаблона; такие типы описаны в главе 26.

## Коллекции

Платформа .NET Framework предлагает разнообразные классы для управления коллекциями элементов. Они включают структуры, основанные на списках и словарях, и работают в сочетании с набором стандартных интерфейсов, которые унифицируют их общие характеристики. Все типы коллекций определены в следующих пространствах имен, описанных в главе 7:

```
System.Collections                  // Необобщенные коллекции
System.Collections.Generic          // Обобщенные коллекции
System.Collections.Specialized      // Строго типизированные коллекции
System.Collections.ObjectModel      // Базовые типы для создания собственных
                                    // коллекций
System.Collections.Concurrent       // Коллекции, безопасные в отношении
                                    // потоков (глава 23)
```

## Запросы

Язык интегрированных запросов (Language Integrated Query – LINQ) появился в версии .NET Framework 3.5. Язык LINQ позволяет выполнять безопасные в отношении типов запросы к локальным и удаленным коллекциям (например, таблицам SQL Server); он описан в главах 8–10. Крупное преимущество языка LINQ в том, что он предоставляет согласованный API-интерфейс запросов для разнообразных предметных областей. Основные типы находятся в перечисленных ниже пространствах имен и являются частью стандарта .NET Standard 2.0:

```
System.Linq               // LINQ to Objects и PLINQ
System.Linq.Expressions   // Для ручного построения выражений
System.Xml.Linq           // LINQ to XML
```

Полная инфраструктура .NET Framework также включает следующие пространства имен, которые будут описаны в разделе "Технологии серверной части" далее в главе:

```
System.Data.Linq          // LINQ to SQL
System.Data.Entity        // LINQ to Entities (Entity Framework)
```

## XML

Язык XML широко применяется внутри .NET Framework, поэтому поддерживается повсеместно. В главе 10 внимание сосредоточено целиком на LINQ to XML – легковесной объектной модели документа XML, которую можно конструировать и опрашивать с помощью LINQ. В главе 11 описана более старая модель W3C DOM, а также высокопроизводительные низкоуровневые классы для чтения/записи плюс поддержка .NET Framework для схем XML, стилевых таблиц и XPath. Ниже перечислены пространства имен, связанные с XML:

```
System.Xml                // XmlReader, XmlWriter и старая модель W3C DOM
System.Xml.Linq           // Объектная модель документа LINQ to XML
System.Xml.Schema         // Поддержка для XSD
System.Xml.Serialization  // Декларативная сериализация XML для типов .NET
System.Xml.XPath          // Язык запросов XPath
System.Xml.Xsl            // Поддержка стилевых таблиц
```

## Диагностика

В главе 13 мы раскроем возможности .NET по регистрации в журнале и утверждениям, а также покажем, как взаимодействовать с другими процессами, выполнять запись в журнал событий Windows и использовать счетчики производительности для проведения мониторинга. Соответствующие типы определены в пространстве имен `System.Diagnostics` и его подпространствах. Средства, специфичные для Windows, не входят в стандарт .NET Standard и доступны только в .NET Framework.

## Параллелизм и асинхронность

Многим современным приложениям в каждый момент времени приходится иметь дело с несколькими действиями. С выходом версии C# 5.0 решение стало проще за счет асинхронных функций и таких высокоуровневых конструкций, как задачи и комбинаторы задач. Все это подробно объясняется в главе 14, которая начинается с рассмотрения основ многопоточности. Типы для работы с потоками и асинхронными операциями находятся в пространствах имен `System.Threading` и `System.Threading.Tasks`.

## Потоки данных и ввод-вывод

Инфраструктура .NET Framework предоставляет потоковую модель для низкоуровневого ввода-вывода. Потоки данных обычно применяются для чтения и записи напрямую в файлы и сетевые подключения и могут соединяться в цепочки либо помещаться внутрь декорированных потоков с целью добавления функциональности сжатия или шифрования. В главе 15 описана потоковая архитектура .NET, а также специфическая поддержка для работы с файлами и каталогами, сжатием, изолированным хранилищем, каналами и файлами, отображенными в память. Тип `Stream` и типы ввода-вывода .NET определены в пространстве имен `System.IO` и его подпространствах, а типы WinRT для файлового ввода-вывода – в пространстве имен `Windows.Storage` и его подпространствах.

## Работа с сетями

С помощью типов из пространства имен `System.Net` можно напрямую работать со стандартными сетевыми протоколами, такими как HTTP, FTP, TCP/IP и SMTP. В главе 16 мы покажем, как взаимодействовать с использованием каждого из упомянутых протоколов, начиная с простых задач вроде загрузки веб-страницы и заканчивая применением TCP/IP для извлечения электронной почты POP3. Ниже перечислены пространства имен, которые будут рассмотрены:

```
System.Net
System.Net.Http      // HttpClient
System.Net.Mail      // Для отправки электронной почты через SMTP
System.Net.Sockets   // TCP, UDP и IP
```

Последние два пространства имен не будут доступны для приложений Windows Store в случае нацеливания на Windows 8/8.1 (WinRT), но доступны для приложений Windows 10 Store (UWP) как часть контракта .NET Standard 2.0. В случае приложений WinRT для отправки электронной почты следует использовать библиотеки от независимых разработчиков, а для работы с сокетами – типы WinRT из пространства имен `Windows.Networking.Sockets`.

## Сериализация

Инфраструктура .NET Framework предлагает несколько систем для сохранения и восстановления объектов в двоичном или текстовом представлении. Такие системы требуются в технологиях распределенных приложений, подобных WCF, Web Services и Remoting, а также применяются для сохранения и восстановления объектов из файлов. В главе 17 мы раскроем три важных механизма сериализации: сериализатор контрактов данных, двоичный сериализатор и сериализатор XML. (В .NET Framework теперь также доступен сериализатор JSON.)

Типы для сериализации находятся в следующих пространствах имен:

```
System.Runtime.Serialization
System.Xml.Serialization
```

## Сборки, рефлексия и атрибуты

Сборки, в которые компилируются программы на C#, состоят из исполняемых инструкций (представленных на промежуточном языке (intermediate language – IL)) и метаданных, описывающих типы, члены и атрибуты программы. С помощью рефлексии можно просматривать метаданные во время выполнения и предпринимать действия вроде динамического вызова методов. Посредством пространства имен `Reflection.Emit` можно конструировать новый код на лету.

В главе 18 мы опишем строение сборок и их подписание, использование глобального кеша сборок (global assembly cache – GAC) и ресурсов, а также распознавание ссылок на файлы. В главе 19 мы раскроем рефлексию и атрибуты – покажем, как инспектировать метаданные, динамически вызывать функции, записывать специальные атрибуты, выпускать новые типы и производить разбор низкоуровневого кода IL. Типы для применения рефлексии и работы со сборками находятся в следующих пространствах имен:

```
System
System.Reflection
System.Reflection.Emit  // Только для .NET Framework
```

## Динамическое программирование

В главе 20 мы рассмотрим несколько паттернов для динамического программирования и работы со средой DLR, которая стала частью CLR, начиная с версии .NET Framework 4.0. Мы покажем, как реализовать паттерн "Посетитель" (Visitor), создавать специальные динамические объекты и взаимодействовать с IronPython. Типы, предназначенные для динамического программирования, находятся в пространстве имен `System.Dynamic`.

## Безопасность

Инфраструктура .NET Framework предоставляет собственный уровень безопасности, позволяя организовать работу в песочнице другим сборкам и себе самой. В главе 21 мы обсудим безопасность доступа кода, безопасность на основе удостоверений и ролей, а также модель прозрачности, введенную в CLR 4.0. Затем мы раскроем криптографические возможности .NET Framework, рассмотрев шифрование, хеширование и защиту данных. Типы для таких целей определены в следующих пространствах имен:

```
System.Security
System.Security.Permissions
System.Security.Policy
System.Security.Cryptography
```

## Расширенная многопоточность

Асинхронные функции в C# значительно облегчают параллельное программирование, поскольку снижают потребность во взаимодействии с низкоуровневыми технологиями. Тем не менее, все еще возникают ситуации, когда нужны сигнальные конструкции, локальное хранилище потока, блокировки чтения/записи и т.д. Данные вопросы

подробно объясняются в главе 22. Типы, связанные с многопоточностью, находятся в пространстве имен `System.Threading`.

## Параллельное программирование

В главе 23 мы рассмотрим библиотеки и типы для работы с многоядерными процессорами, включая API-интерфейсы для реализации параллелизма задач, императивного параллелизма данных и функционального параллелизма (PLINQ).

## Домены приложений

Среда CLR предоставляет дополнительный уровень изоляции внутри процесса, который называется *доменом приложения*. В главе 24 мы исследуем свойства домена приложения, с которыми можно взаимодействовать, и покажем, как создавать и использовать дополнительные домены приложений в рамках одного и того же процесса для таких целей, как модульное тестирование. Мы также объясним, каким образом применять технологию Remoting для взаимодействия с доменами приложений.

Создание отдельных доменов приложений не является частью стандарта .NET Standard 2.0, хотя можно взаимодействовать с текущим доменом через класс `AppDomain` из пространства имен `System`.

## Собственная возможность взаимодействия и возможность взаимодействия с COM

Существует возможность взаимодействия с собственным кодом, а также с кодом COM. Собственная возможность взаимодействия позволяет вызывать функции из неуправляемых DLL-библиотек, регистрировать обратные вызовы, отображать структуры данных и работать с собственными типами данных. Возможность взаимодействия с COM позволяет обращаться к типам COM и открывать для COM доступ к типам .NET. Типы, поддерживающие такую функциональность, определены в пространстве имен `System.Runtime.InteropServices` и рассматриваются в главе 25.

# Прикладные технологии

## Технологии пользовательских интерфейсов

Приложения, основанные на пользовательском интерфейсе, можно разделить на две категории: *тонкий клиент*, равнозначный веб-сайту, и *обогащенный клиент*, представляющий собой программу, которую конечный пользователь должен загрузить и установить на компьютере или на мобильном устройстве.

Для разработки приложений тонких клиентов платформа .NET предлагает инфраструктуры ASP.NET и ASP.NET Core.

Для построения приложений обогащенных клиентов, ориентированных на рабочий стол Windows 7/8/10, платформа .NET предоставляет API-интерфейсы WPF и Windows Forms. Для создания приложений обогащенных клиентов, нацеленных на iOS, Android и Windows Phone, имеется инфраструктура Xamarin, а для написания приложений обогащенных клиентов Windows Store, функционирующих на настольных компьютерах и устройствах Windows 10, доступна платформа UWP (см. табл. 1.1 в главе 1).

Наконец, существует гибридная технология под названием Silverlight, с которой практически перестали работать после выхода HTML5.

## ASP.NET

Приложения, написанные с использованием ASP.NET, размещаются на сервере Windows IIS и могут быть доступны из любого веб-браузера. Ниже перечислены преимущества ASP.NET по сравнению с технологиями обогащенных клиентов.

- Отсутствует потребность в развертывании на клиентской стороне.
- Клиенты могут функционировать на платформах, отличных от Windows.
- Легко развертывать обновления.

Кроме того, поскольку бо́льшая часть кода, написанного в приложении ASP.NET, выполняется на сервере, уровень доступа к данным проектируется для функционирования в том же самом домене приложения — без ограничения безопасности или масштабируемости. Напротив, обогащенный клиент, который делает то же самое, обычно не настолько безопасный или масштабируемый. (В случае обогащенного клиента решение предусматривает создание *среднего уровня* между клиентом и базой данных. Средний уровень выполняется на удаленном сервере приложений (часто вместе с сервером базы данных) и взаимодействует с обогащенными клиентами через WCF, Web Services или Remoting.)

При написании своих веб-страниц вы можете выбирать между традиционным API-интерфейсом Web Forms и более новым API-интерфейсом MVC (Model-View-Controller — модель-представление-контроллер). Оба они построены на основе инфраструктуры ASP.NET. Технология Web Forms была частью .NET Framework с самого начала, а MVC реализована намного позже как реакция на успех Ruby on Rails и MonoRail. В целом инфраструктура MVC обеспечивает лучшую программную абстракцию по сравнению с Web Forms; она также позволяет иметь больший контроль над генерируемой HTML-разметкой. Единственное, в чем MVC проигрывает Web Forms — визуальный конструктор, что сохраняет Web Forms хорошим средством для построения веб-страниц с преимущественно статическим содержимым.

Ограничения ASP.NET в значительной степени отражают общие ограничения систем тонких клиентов.

- Несмотря на то что веб-браузер способен предложить насыщенный мощный интерфейс с помощью HTML5 и AJAX, в плане возможностей и производительности он по-прежнему уступает собственному API-интерфейсу обогащенного клиента.
- Поддержка состояния на стороне клиента — или от имени клиента — может быть громоздкой.

Типы, предназначенные для написания приложений ASP.NET, находятся в пространстве имен `System.Web.UI` и его подпространствах; они упакованы в сборку *`System.Web.dll`*. Инфраструктура ASP.NET доступна через NuGet.

## ASP.NET Core

Относительно недавно добавленная инфраструктура ASP.NET Core похожа на ASP.NET, но функционирует в средах .NET Framework и .NET Core (делая возможным межплатформенное развертывание). Инфраструктура ASP.NET Core отличается легковесной модульной архитектурой, обладает способностью саморазмещения в специальном процессе и регламентируется лицензией для программного обеспечения с открытым кодом. В отличие от своих предшественников инфраструктура ASP.NET Core не зависит от `System.Web` и лишена исторического багажа Web Forms. Она особенно хорошо подходит для микросервисов и развертывания внутри контейнеров.

### Windows Presentation Foundation (WPF)

Инфраструктура WPF была введена в .NET Framework 3.0 для написания приложений обогащенных клиентов. Ниже перечислены преимущества инфраструктуры WPF по сравнению с Windows Forms.

- Она поддерживает развитую графику, включая произвольные трансформации, трехмерную визуализацию, мультимедиа-возможности и подлинную прозрачность. Оформление поддерживается через стили и шаблоны.
- Основная единица измерения не базируется на пикселях, поэтому приложения корректно отображаются при любой настройке DPI (dots per inch – точек на дюйм).
- Она располагает обширной и гибкой поддержкой динамической компоновки, означающей возможность локализации приложения без опасности того, что элементы будут перекрывать друг друга.
- Визуализация применяет технологию DirectX и отличается высокой скоростью, извлекая крупные преимущества от аппаратного ускорения графики.
- Она предлагает надежную привязку к данным.
- Пользовательские интерфейсы могут быть описаны декларативно в XAML-файлах, которые поддерживаются независимо от файлов отделенного кода, что помогает отделить внешний вид от функциональности.

Однако размеры и сложность WPF обусловливают крутую кривую обучения.

Типы, предназначенные для написания WPF-приложений, находятся в пространстве имен `System.Windows` и всех его подпространствах за исключением `System.Windows.Forms`.

### Windows Forms

Windows Forms – это API-интерфейс обогащенного клиента, который является ровесником самой платформы .NET Framework. По сравнению с WPF она является относительно простой технологией, которая предлагает большинство возможностей, необходимых во время разработки типового Windows-приложения. Она также играет важную роль в сопровождении унаследованных приложений. Тем не менее, в сравнении с WPF инфраструктура Windows Forms обладает рядом недостатков.

- Позиции и размеры элементов управления задаются в пикселях, что приводит к риску некорректного отображения приложений на клиентах с настройками DPI, отличающимися от таких настроек у разработчика (хотя в версии .NET Framework 4.7 ситуация несколько улучшилась).
- Для рисования нестандартных элементов управления используется API-интерфейс GDI+, который вопреки достаточно высокой гибкости медленно визуализирует крупные области (и без двойной буферизации может вызывать мерцание).
- Элементы управления лишены подлинной прозрачности.
- Большинство элементов управления не поддерживают компоновку. Например, поместить элемент управления изображением внутрь заголовка элемента управления вкладкой не удастся. Настройка списковых представлений и полей с раскрывающимися списками отнимает много времени и сил.
- Трудно добиться надежной работы динамической компоновки.

Последний пункт является веской причиной отдавать предпочтение WPF перед Windows Forms, даже если разрабатывается бизнес-приложение, которому требуется просто пользовательский интерфейс, а не учет "поведенческих особенностей пользователей". Элементы компоновки в WPF, подобные `Grid`, упрощают организацию меток и текстовых полей таким образом, что они будут всегда выровненными – даже при смене языка локализации – без запутанной логики и какого-либо мерцания. Кроме того, не придется приводить все к наименьшему общему знаменателю в смысле экранного разрешения – элементы компоновки WPF изначально проектировались с поддержкой изменения размеров.

В качестве положительного момента следует отметить, что инфраструктура Windows Forms относительно проста в изучении и все еще поддерживается в многочисленных элементах управления от независимых разработчиков.

Типы Windows Forms находятся в пространствах имен `System.Windows.Forms` (сборка *`System.Windows.Forms.dll`*) и `System.Drawing` (сборка *`System.Drawing.dll`*). Последнее пространство имен также содержит типы GDI+ для рисования специальных элементов управления.

### Xamarin

Инфраструктура Xamarin, которой теперь владеет Microsoft, позволяет писать мобильные приложения на языке C#, которые ориентированы на iOS и Android, а также на Windows Phone. Будучи межплатформенной, она функционирует не только под управлением .NET Framework, но и в своей собственной среде (производной от среды с открытым кодом Mono). За дополнительной информацией обращайтесь на веб-сайт `https://www.xamarin.com`.

### UWP (Universal Windows Platform)

Универсальная платформа Windows (UWP) предназначена для разработки приложений, ориентированных на настольные компьютеры и устройства Windows 10, которые распространяются через Windows Store. Ее API-интерфейс обогащенного клиента, на который большое влияние оказала инфраструктура WPF, рассчитан на построение пользовательских интерфейсов, управляемых касаниями, и для компоновки применяет язык XAML. Типы находятся в пространствах имен `Windows.UI` и `Windows.UI.Xaml`.

### Silverlight

Платформа Silverlight также отделена от .NET Framework и позволяет создавать графические пользовательские интерфейсы, которые функционируют в веб-браузере, во многом похоже на Macromedia Flash. С развитием HTML5 в Microsoft перестали уделять внимание Silverlight.

## Технологии серверной части

### ADO.NET

ADO.NET – это управляемый API-интерфейс доступа к данным. Хотя название включает наименование используемой в 1990-х годах технологии ADO (ActiveX Data Objects – объекты данных ActiveX), ADO.NET – совершенно другая технология. Она содержит два основных низкоуровневых компонента.

#### Уровень поставщиков

Модель поставщиков определяет общие классы и интерфейсы для низкоуровневого доступа к поставщикам баз данных. Такие интерфейсы состоят из подключений, команд, адаптеров и средств чтения (однонаправленных курсоров, предназначенных только для чтения базы данных). Инфраструктура .NET Framework поставляется с собственной поддержкой Microsoft SQL Server, но доступно множество драйверов от независимых разработчиков для других баз данных.

#### Модель `DataSet`

`DataSet` — это структурированный кеш данных. Он напоминает примитивную базу данных в памяти, которая определяет такие SQL-конструкции, как таблицы, строки, столбцы, отношения и представления. За счет программирования с участием кеша данных можно сократить количество обращений к серверу, улучшая показатели масштабируемости сервера и отзывчивости пользовательского интерфейса обогащенного клиента. Объекты `DataSet` поддерживают сериализацию и могут передаваться по сети между клиентскими и серверными приложениями.

Поверх уровня поставщиков находятся три API-интерфейса, которые предлагают возможность запрашивать базы данных с помощью LINQ:

- Entity Framework (только .NET Framework);
- Entity Framework Core (.NET Framework и .NET Core);
- LINQ to SQL (только .NET Framework).

Все три технологии включают *объектно-реляционные отображатели* (object/relational mapper — ORM), которые автоматически отображают объекты (основанные на определяемых вами классах) на строки в базе данных. Это позволяет запрашивать такие объекты с применением LINQ (вместо написания SQL-операторов `SELECT`) и обновлять их без написания вручную SQL-операторов `INSERT/DELETE/UPDATE`. В результате сокращается объем кода внутри уровня доступа к данным в приложении (особенно вспомогательного кода) и обеспечивается строгая статическая безопасность типов. Указанные технологии также устраняют необходимость в наличии `DataSet` как вместилищ данных, хотя объекты `DataSet` по-прежнему предлагают уникальную возможность по хранению и сериализации изменений состояния (то, что особенно полезно в многоуровневых приложениях). В сочетании с `DataSet` можно использовать Entity Framework или LINQ to SQL, хотя такой подход несколько грубый ввиду неуклюжести самих `DataSet`. Другими словами, пока еще не существует прямолинейного готового решения для написания *n*-уровневых приложений с ORM от Microsoft.

Технология LINQ to SQL проще и быстрее Entity Framework, к тому же исторически производит лучший SQL-код (хотя Entity Framework улучшается благодаря многочисленным обновлениям). Технология Entity Framework более гибкая в том, что позволяет создавать точные отображения между базой данных и запрашиваемыми классами (Entity Data Model — модель сущностных данных), и предлагает модель, которая делает возможной независимую поддержку для баз данных, отличающихся от SQL Server.

Инфраструктура Entity Framework Core (EF Core) — это переписанная инфраструктура Entity Framework с более простым проектным решением, навеянным LINQ to SQL. Она отказывается от сложной модели сущностных данных и функционирует под управлением .NET Framework и .NET Core.

Стандарт .NET Standard 2.0 включает общие интерфейсы из уровня поставщиков, а также объекты `DataSet`, но исключает типы, специфичные для SQL Servers, и объектно-реляционные отображатели.

### Windows Workflow (только .NET Framework)

Windows Workflow – это инфраструктура для моделирования и управления потенциально длительно выполняющимися бизнес-процессами. Инфраструктура Windows Workflow ориентирована на стандартную библиотеку времени выполнения, обеспечивая согласованность и возможность взаимодействия. Кроме того, она помогает сократить объем кодирования для динамически управляемых деревьев принятия решений.

Инфраструктура Windows Workflow не является строго серверной технологией – ее можно применять где угодно (например, организовать поток страницы в пользовательском интерфейсе).

Первоначально инфраструктура Windows Workflow появилась в версии .NET Framework 3.0, а ее типы были определены в пространстве имен `System.WorkFlow`. В .NET Framework 4.0 она была полностью пересмотрена; новые типы теперь находятся в пространстве имен `System.Activities`.

### COM+ и MSMQ (только .NET Framework)

Инфраструктура .NET Framework позволяет взаимодействовать с COM+ для таких служб, как распределенные транзакции, через типы из пространства имен `System.EnterpriseServices`. Она также поддерживает технологию MSMQ (Microsoft Message Queuing – организация очереди сообщений Microsoft) для асинхронного однонаправленного обмена сообщениями посредством типов из пространства имен `System.Messaging`.

## Технологии распределенных систем

### Windows Communication Foundation (WCF)

WCF представляет собой сложную инфраструктуру для коммуникаций, которая появилась в версии .NET Framework 3.0. Она является гибкой и достаточно конфигурируемой для того, чтобы сделать своих предшественников – Remoting и Web Services (.ASMX) – *по большей части* излишними.

WCF, Remoting и Web Services похожи между собой в том, что все они реализуют описанную ниже базовую модель коммуникаций между клиентским и серверным приложениями.

- На стороне сервера вы указываете, какие методы могут быть вызваны удаленными клиентскими приложениями.
- На стороне клиента вы указываете или выводите *сигнатуры* серверных методов, которые должны вызываться.
- На сторонах сервера и клиента вы выбираете транспортный и коммуникационный протокол (в WCF это делается посредством *привязки*).
- Клиент устанавливает подключение к серверу.
- Клиент вызывает удаленный метод, который прозрачно выполняется на сервере.

Инфраструктура WCF еще более развязывает клиент и сервер через контракты служб и контракты данных. Концептуально вместо того, чтобы напрямую вызывать удаленный *метод*, клиент отправляет сообщение (XML или двоичное) конечной точке

удаленной *службы*. Одно из преимуществ такой развязки заключается в том, что клиенты не имеют какой-либо зависимости от платформы .NET или от любых патентованных коммуникационных протоколов.

Инфраструктура WCF исключительно конфигурируема и предлагает широкую поддержку для стандартизированных протоколов обмена сообщениями, основанных на SOAP (Simple Object Access Protocol – простой протокол доступа к объектам), в том числе расширения WS-* для безопасности. В результате появляется возможность взаимодействовать с участниками, выполняющими другое программное обеспечение – вероятно, на разных платформах – и в то же время по-прежнему поддерживать расширенные средства вроде шифрования. Однако на практике сложность этих протоколов ограничивает их адаптацию другими платформами, и в настоящий момент наилучшим вариантом для обмена сообщениями с возможностью взаимодействия является архитектурный стиль REST поверх HTTP, который в Microsoft поддерживается посредством уровня Web API на основе ASP.NET.

Тем не менее, для коммуникации между системами .NET инфраструктура WCF предлагает более развитую сериализацию и улучшенные инструменты, чем доступные с API-интерфейсами REST. Она также потенциально быстрее, не привязана к HTTP и может использовать двоичную сериализацию.

Типы, предназначенные для организации коммуникаций с помощью WCF, находятся в пространстве имен `System.ServiceModel` и его подпространствах.

### Web API

Инфраструктура Web API функционирует поверх ASP.NET/ASP.NET Core и архитектурно подобна API-интерфейсу MVC от Microsoft за исключением того, что она спроектирована для открытия доступа к службам и данным, а не веб-страницам. Преимущество инфраструктуры Web API перед WCF связано с тем, что она позволяет следовать популярным соглашениям REST поверх HTTP, предлагая несложное взаимодействие с широчайшим диапазоном платформ.

Внутренне реализации REST проще протоколов SOAP и WS-*, на которые WCF полагается в плане возможности взаимодействия. С точки зрения архитектуры API-интерфейсы REST более элегантны для систем со слабой связностью, построены на основе фактических стандартов и великолепно задействуют то, что уже предоставляет протокол HTTP.

### Remoting и .ASMX Web Services (только .NET Framework)

Remoting и .ASMX Web Services являются предшественниками WCF. С появлением WCF технология Remoting стала почти излишней, а .ASMX Web Services – излишней целиком и полностью.

Оставшаяся ниша Remoting касается коммуникаций между доменами приложений внутри одного и того же процесса (глава 24). Технология Remoting ориентирована на приложения с сильной связностью. Типичным примером может служить ситуация, когда клиент и сервер представляют собой приложения .NET, написанные одной компанией (или компаниями, разделяющими общие сборки). Коммуникации обычно предусматривают обмен потенциально сложными специальными объектами .NET, которые инфраструктура Remoting сериализирует и десериализирует без необходимости во внешнем вмешательстве.

Типы для Remoting находятся в пространстве имен `System.Runtime.Remoting` или его подпространствах, а типы для Web Services – в пространстве имен `System.Web.Services`.

# 6

# Основы .NET Framework

Многие ключевые возможности, необходимые во время программирования, предоставляются не языком C#, а типами в .NET Framework. В этой главе мы раскроем роль .NET Framework при решении фундаментальных задач программирования, таких как виртуальное сравнение эквивалентности, сравнение порядка и преобразование типов. Мы также рассмотрим базовые типы .NET Framework, подобные `String`, `DateTime` и `Enum`.

Описанные здесь типы находятся в пространстве имен `System` со следующими исключениями:

- `StringBuilder` определен в пространстве имен `System.Text`, т.к. относится к типам для *кодировок текста*;
- `CultureInfo` и связанные с ним типы определены в пространстве имен `System.Globalization`;
- `XmlConvert` определен в пространстве имен `System.Xml`.

## Обработка строк и текста

### Тип `char`

Тип `char` в C# представляет одиночный символ Unicode и является псевдонимом структуры `System.Char`. В главе 2 было показано, как выражать литералы `char`. Например:

```
char c = 'A';
char newLine = '\n';
```

В структуре `System.Char` определен набор статических методов для работы с символами, в том числе `ToUpper`, `ToLower` и `IsWhiteSpace`. Их можно вызывать либо через тип `System.Char`, либо через его псевдоним `char`:

```
Console.WriteLine (System.Char.ToUpper ('c'));  // C
Console.WriteLine (char.IsWhiteSpace ('\t'));   // True
```

Методы ToUpper и ToLower учитывают локаль конечного пользователя, что может приводить к неуловимым ошибкам. Следующее выражение дает false для турецкой локали:

```
char.ToUpper ('i') == 'I'
```

поскольку в данном случае char.ToUpper ('i') равно 'İ' (обратите внимание на точку в верхней части буквы). Чтобы избежать проблем подобного рода, тип System.Char (и System.String) также предлагает независимые от культуры версии методов ToUpper и ToLower, имена которых завершаются словом *Invariant*. В них всегда применяется правила культуры английского языка:

```
Console.WriteLine (char.ToUpperInvariant ('i')); // I
```

Это является сокращением для такого кода:

```
Console.WriteLine (char.ToUpper ('i', CultureInfo.InvariantCulture));
```

Дополнительные сведения о локалях и культурах можно найти в разделе "Форматирование и разбор" далее в этой главе.

Большинство оставшихся статических методов типа char имеет отношение к категоризации символов; они перечислены в табл. 6.1.

**Таблица 6.1. Статические методы для категоризации символов**

| Статический метод | Включает символы | Включает категории Unicode |
|---|---|---|
| IsLetter | A–Z, a–z и все буквы из других алфавитов | UpperCaseLetter<br>LowerCaseLetter<br>TitleCaseLetter<br>ModifierLetter<br>OtherLetter |
| IsUpper | Буквы в верхнем регистре | UpperCaseLetter |
| IsLower | Буквы в нижнем регистре | LowerCaseLetter |
| IsDigit | 0–9 и все цифры из других алфавитов | DecimalDigitNumber |
| IsLetterOrDigit | Буквы и цифры | IsLetter, IsDigit |
| IsNumber | Все цифры, а также дроби Unicode и числовые символы латинского набора | DecimalDigitNumber<br>LetterNumber<br>OtherNumber |
| IsSeparator | Пробел и все символы разделителей Unicode | LineSeparator<br>ParagraphSeparator |
| IsWhiteSpace | Все разделители, а также \n, \r, \t, \f и \v | LineSeparator<br>ParagraphSeparator |
| IsPunctuation | Символы, используемые для пунктуации в латинском и других алфавитах | DashPunctuation<br>ConnectorPunctuation<br>InitialQuotePunctuation<br>FinalQuotePunctuation |
| IsSymbol | Большинство других печатаемых символов | MathSymbol<br>ModifierSymbol<br>OtherSymbol |
| IsControl | Непечатаемые "управляющие" символы с кодами меньше 0x20, такие как \r, \n, \t, \0, и символы с кодами между 0x7F и 0x9A | – |

Для более детальной категоризации тип char предоставляет статический метод по имени GetUnicodeCategory; он возвращает перечисление UnicodeCategory, члены которого были показаны в правой колонке табл. 6.1.

При явном приведении целочисленного значения вполне возможно получить значение char, выходящее за пределы выделенного набора Unicode. Для проверки допустимости символа необходимо вызвать метод char.GetUnicodeCategory: если результатом окажется UnicodeCategory.OtherNotAssigned, то символ является недопустимым.

Значение char имеет ширину 16 битов, которой достаточно для представления любого символа Unicode в *базовой многоязыковой плоскости*. Чтобы выйти за ее пределы, потребуется применять суррогатные пары: мы опишем необходимые методы в разделе "Кодировка текста и Unicode" далее в главе.

## Тип string

Тип string (псевдоним класса System.String) в C# является неизменяемой последовательностью символов. В главе 2 мы объяснили, как выражать строковые литералы, выполнять сравнения эквивалентности и осуществлять конкатенацию двух строк. В текущем разделе мы рассмотрим остальные функции для работы со строками, которые доступны через статические члены и члены экземпляра класса System.String.

### Конструирование строк

Простейший способ конструирования строки предусматривает присваивание литерала, как было показано в главе 2:

```
string s1 = "Hello";
string s2 = "First Line\r\nSecond Line";
string s3 = @"\\server\fileshare\helloworld.cs";
```

Чтобы создать повторяющуюся последовательность символов, можно использовать конструктор string:

```
Console.Write (new string ('*', 10)); // **********
```

Строку можно также сконструировать из массива char. Метод ToCharArray выполняет обратное действие:

```
char[] ca = "Hello".ToCharArray();
string s = new string (ca);              // s = "Hello"
```

Конструктор типа string имеет перегруженные версии, которые принимают разнообразные (небезопасные) типы указателей и предназначены для создания строк из таких типов, как char*.

### Строки null и пустые строки

Пустая строка имеет нулевую длину. Чтобы создать пустую строку, можно применить либо литерал, либо статическое поле string.Empty; для проверки, пуста ли строка, можно либо выполнить сравнение эквивалентности, либо просмотреть свойство Length строки:

```
string empty = "";
Console.WriteLine (empty == "");            // True
Console.WriteLine (empty == string.Empty);  // True
Console.WriteLine (empty.Length == 0);      // True
```

Поскольку строки являются ссылочными типами, они также могут быть null:

```
string nullString = null;
Console.WriteLine (nullString == null);          // True
Console.WriteLine (nullString == "");            // False
Console.WriteLine (nullString.Length == 0);      // Генерируется исключение
                                                 // NullReferenceException
```

Статический метод string.IsNullOrEmpty является удобным сокращением для проверки, равна ли заданная строка null или является пустой.

### Доступ к символам внутри строки

Индексатор строки возвращает одиночный символ по указанному индексу. Как и во всех функциях для работы со строками, индекс начинается с нуля:

```
string str  = "abcde";
char letter = str[1];  // letter == 'b'
```

Тип string также реализует интерфейс IEnumerable<char>, так что по символам строки можно проходить с помощью foreach:

```
foreach (char c in "123") Console.Write (c + ",");         // 1,2,3,
```

### Поиск внутри строк

К простейшим методам поиска внутри строк относятся StartsWith, EndsWith и Contains. Все они возвращают true или false:

```
Console.WriteLine ("quick brown fox".EndsWith ("fox"));   // True
Console.WriteLine ("quick brown fox".Contains ("brown")); // True
```

Методы StartsWith и EndsWith перегружены, чтобы позволить указывать перечисление StringComparison или объект CultureInfo для управления чувствительностью к регистру символов и культуре (см. раздел "Ординальное сравнение или сравнение, чувствительное к культуре" далее в главе). По умолчанию выполняется сопоставление, чувствительное к культуре, с использованием правил, которые применимы к текущей (локализованной) культуре. В следующем случае производится поиск, нечувствительный к регистру символов, с использованием правил, *не зависящих* от культуры:

```
"abcdef".StartsWith ("aBc", StringComparison.InvariantCultureIgnoreCase)
```

Метод Contains не имеет такой перегруженной версии, хотя того же результата можно добиться с помощью метода IndexOf.

Метод IndexOf более мощный: он возвращает позицию первого вхождения заданного символа или подстроки (или -1, если символ или подстрока не найдена):

```
Console.WriteLine ("abcde".IndexOf ("cd"));  // 2
```

Метод IndexOf также перегружен для приема startPosition (индекса, с которого должен начинаться поиск) и перечисления StringComparison:

```
Console.WriteLine ("abcde abcde".IndexOf ("CD", 6,
                   StringComparison.CurrentCultureIgnoreCase));  // 8
```

Метод LastIndexOf похож на IndexOf, но перемещается по строке в обратном направлении.

Метод IndexOfAny возвращает позицию первого вхождения любого символа из множества символов:

```
Console.Write ("ab,cd ef".IndexOfAny (new char[] {' ', ','} ));          // 2
Console.Write ("pas5w0rd".IndexOfAny ("0123456789".ToCharArray() ));    // 3
```

Метод `LastIndexOfAny` делает то же самое, но в обратном направлении.

### Манипулирование строками

Поскольку класс `String` неизменяемый, все методы, которые "манипулируют" строкой, возвращают новую строку, оставляя исходную незатронутой (то же самое происходит при повторном присваивании строковой переменной).

Метод `Substring` извлекает порцию строки:

```
string left3 = "12345".Substring (0, 3);          // left3 = "123";
string mid3  = "12345".Substring (1, 3);          // mid3 = "234";
```

Если длина не указана, тогда извлекается порция до самого конца строки:

```
string end3  = "12345".Substring (2);             // end3 = "345";
```

Методы `Insert` и `Remove` вставляют либо удаляют символы в указанной позиции:

```
string s1 = "helloworld".Insert (5, ", ");        // s1 = "hello, world"
string s2 = s1.Remove (5, 2);                     // s2 = "helloworld";
```

Методы `PadLeft` и `PadRight` дополняют строку до заданной длины слева или справа заданным символом (или пробелом, если символ не указан):

```
Console.WriteLine ("12345".PadLeft (9, '*'));     // ****12345
Console.WriteLine ("12345".PadLeft (9));          //     12345
```

Если входная строка длиннее, чем длина для дополнения, то исходная строка возвращается без изменений.

Методы `TrimStart` и `TrimEnd` удаляют указанные символы из начала или конца строки; метод `Trim` делает то и другое. По умолчанию эти функции удаляют пробельные символы (включающие пробелы, табуляции, символы новой строки и их вариации в Unicode):

```
Console.WriteLine ("  abc \t\r\n ".Trim().Length);   // 3
```

Метод `Replace` заменяет все (неперекрывающиеся) вхождения заданного символа или подстроки:

```
Console.WriteLine ("to be done".Replace (" ", " | ") );    // to | be | done
Console.WriteLine ("to be done".Replace (" ", "")    );    // tobedone
```

Методы `ToUpper` и `ToLower` возвращают версии входной строки в верхнем и нижнем регистре символов. По умолчанию они учитывают текущие языковые настройки у пользователя; методы `ToUpperInvariant` и `ToLowerInvariant` всегда применяют правила, принятые для английского алфавита.

### Разделение и объединение строк

Метод `Split` разделяет строку на порции:

```
string[] words = "The quick brown fox".Split();

foreach (string word in words)
  Console.Write (word + "|");  // The|quick|brown|fox|
```

По умолчанию в качестве разделителей метод `Split` использует пробельные символы; также имеется его перегруженная версия, принимающая массив `params` разделителей `char` или `string`. Кроме того, метод `Split` дополнительно принимает

перечисление StringSplitOptions, в котором предусмотрен вариант для удаления пустых элементов: он полезен, когда слова в строке разделяются несколькими разделителями.

Статический метод Join выполняет действие, противоположное Split. Он требует указания разделителя и строкового массива:

```
string[] words = "The quick brown fox".Split();
string together = string.Join (" ", words);  // The quick brown fox
```

Статический метод Concat похож на Join, но принимает только строковый массив params и не задействует разделители. Метод Concat в точности эквивалентен операции + (на самом деле компилятор транслирует операцию + в вызов Concat):

```
string sentence     = string.Concat ("The", " quick", " brown", " fox");
string sameSentence = "The" + " quick" + " brown" + " fox";
```

### Метод String.Format и смешанные форматные строки

Статический метод Format предлагает удобный способ для построения строк, которые содержат в себе переменные. Эти встроенные переменные (или значения) могут относиться к любому типу; метод Format просто вызывает на них ToString.

Главная строка, включающая встроенные переменные, называется *смешанной форматной строкой*. При вызове методу String.Format предоставляется такая смешанная форматная строка, а за ней по очереди все встроенные переменные. Например:

```
string composite = "It's {0} degrees in {1} on this {2} morning";
string s = string.Format (composite, 35, "Perth", DateTime.Now.DayOfWeek);

// s == "It's 35 degrees in Perth on this Friday morning"
```

Начиная с версии C# 6, для получения тех же результатов можно применять интерполированные строковые литералы (см. раздел "Строковый тип" в главе 2). Достаточно просто предварить строку символом $ и поместить выражения в фигурные скобки:

```
string s = $"It's hot this {DateTime.Now.DayOfWeek} morning";
```

Каждое число в фигурных скобках называется *форматным элементом*. Число соответствует позиции аргумента и за ним может дополнительно следовать:

- запятая и *минимальная ширина*;
- двоеточие и *форматная строка*.

Минимальная ширина удобна для выравнивания колонок. Если значение отрицательное, тогда данные выравниваются влево, а иначе — вправо. Например:

```
string composite = "Name={0,-20} Credit Limit={1,15:C}";

Console.WriteLine (string.Format (composite, "Mary", 500));
Console.WriteLine (string.Format (composite, "Elizabeth", 20000));
```

Вот как выглядит результат:

```
Name=Mary            Credit Limit=        $500.00
Name=Elizabeth       Credit Limit=     $20,000.00
```

Ниже приведен эквивалентный код, в котором метод string.Format не применяется:

```
string s = "Name=" + "Mary".PadRight (20) +
           " Credit Limit=" + 500.ToString ("C").PadLeft (15);
```

Значение кредитного лимита (`Credit Limit`) форматируется как денежное посредством форматной строки `"C"`. Форматные строки более подробно рассматриваются в разделе "Форматирование и разбор" далее в главе.

## Сравнение строк

При сравнении двух значений в .NET Framework проводится различие между концепциями *сравнения эквивалентности* и *сравнения порядка*. Сравнение эквивалентности проверяет, являются ли два экземпляра семантически одинаковыми; сравнение порядка выясняет, какой из двух экземпляров (если есть) будет следовать первым в случае расположения их по возрастанию или убыванию.

Сравнение эквивалентности не является *подмножеством* сравнения порядка; две системы сравнения имеют разное предназначение. Вполне допустимо иметь, скажем, два неравных значения в одной и той же порядковой позиции. Мы продолжим данную тему в разделе "Сравнение эквивалентности" позже в главе.

Для сравнения эквивалентности строк можно использовать операцию == или один из методов `Equals` типа `string`. Последние более универсальны, потому что позволяют указывать такие варианты, как нечувствительность к регистру символов.

Другое отличие связано с тем, что операция == не работает надежно со строками, если переменные приведены к типу `object`. Мы объясним это в разделе "Сравнение эквивалентности" далее в главе.

Для сравнения порядка строк можно применять либо метод экземпляра `CompareTo`, либо статические методы `Compare` и `CompareOrdinal`: они возвращают положительное или отрицательное число либо ноль – в зависимости от того, находится первое значение до, после или рядом со вторым.

Прежде чем углубляться в детали каждого вида сравнения, необходимо изучить лежащие в основе .NET алгоритмы сравнения строк.

### Ординальное сравнение или сравнение, чувствительное к культуре

При сравнении строк используются два базовых алгоритма: алгоритм *ординального* сравнения и алгоритм сравнения, *чувствительного к культуре*. В случае ординального сравнения символы интерпретируются просто как числа (согласно своим числовым кодам Unicode), а в случае сравнения, чувствительного к культуре, символы интерпретируются со ссылкой на конкретный словарь. Существуют две специальные культуры: "текущая культура", которая основана на настройках, получаемых из панели управления компьютера, и "инвариантная культура", которая является одной и той же на всех компьютерах (и полностью соответствует американской культуре).

Для сравнения эквивалентности удобны оба алгоритма. Однако при упорядочивании сравнение, чувствительное к культуре, почти всегда предпочтительнее: для алфавитного упорядочения строк необходим алфавит. Ординальное сравнение полагается на числовые коды Unicode, которые, так уж случилось, выстраивают английские символы в алфавитном порядке – но не в точности так, как можно было бы ожидать. Например, предполагая чувствительность к регистру, рассмотрим строки "Atom", "atom" и "Zamia". В случае инвариантной культуры они располагаются в следующем порядке:

```
"Atom", "atom", "Zamia"
```

Тем не менее, при ординальном сравнении результат выглядит так:

```
"Atom", "Zamia", "atom"
```

Причина в том, что инвариантная культура инкапсулирует алфавит, в котором символы в верхнем регистре находятся рядом со своими двойниками в нижнем регистре (AaBbCcDd...). Но при ординальном сравнении сначала идут все символы в верхнем регистре, а затем – все символы в нижнем регистре (A...Z, a...z). В сущности, производится возврат к набору символов ASCII, появившемуся в 1960-х годах.

### Сравнение эквивалентности строк

Несмотря на ограничения ординального сравнения, операция == в типе `string` всегда выполняет *ординальное сравнение, чувствительное к регистру*. То же самое касается версии экземпляра метода `string.Equals` в случае вызова без параметров; это определяет "стандартное" поведение сравнения эквивалентности для типа `string`.

Алгоритм ординального сравнения выбран для функций == и `Equals` типа `string` из-за того, что он высокоэффективен и *детерминирован*. Сравнение эквивалентности строк считается фундаментальной операцией и выполняется гораздо чаще, чем сравнение порядка.

"Строгое" понятие эквивалентности также согласуется с общим применением операции ==.

Следующие методы позволяют выполнять сравнение с учетом культуры или сравнение, нечувствительное к регистру:

```
public bool Equals(string value, StringComparison comparisonType);
public static bool Equals (string a, string b,
                           StringComparison comparisonType);
```

Статическая версия полезна тем, что она работает в случае, когда одна или обе строки равны `null`. Перечисление `StringComparison` определено так, как показано ниже:

```
public enum StringComparison
{
  CurrentCulture,             // Чувствительное к регистру
  CurrentCultureIgnoreCase,
  InvariantCulture,           // Чувствительное к регистру
  InvariantCultureIgnoreCase,
  Ordinal,                    // Чувствительное к регистру
  OrdinalIgnoreCase
}
```

Вот примеры:

```
Console.WriteLine (string.Equals ("foo", "FOO",
                   StringComparison.OrdinalIgnoreCase));  // True
Console.WriteLine ("ǔ" == "ǔ");                           // False
Console.WriteLine (string.Equals ("ǔ", "ǔ",
                   StringComparison.CurrentCulture));     // ?
```

(Результат в третьем операторе определяется текущими настройками языка на компьютере.)

### Сравнение порядка строк

Метод экземпляра `CompareTo` класса `String` выполняет чувствительное к культуре и регистру сравнение порядка. В отличие от операции == метод `CompareTo` не использует ординальное сравнение: для упорядочивания намного более полезен алгоритм сравнения, чувствительного к культуре.

Вот определение метода `CompareTo`:

```
public int CompareTo (string strB);
```

Метод экземпляра `CompareTo` реализует обобщенный интерфейс `IComparable`, который является стандартным протоколом сравнения, повсеместно применяемым в .NET Framework. Это значит, что метод `CompareTo` типа `string` определяет строки со стандартным поведением упорядочивания в таких реализациях, как сортированные коллекции, например. За дополнительной информацией по `IComparable` обращайтесь в раздел "Сравнение порядка" далее в главе.

Для других видов сравнения можно вызывать статические методы `Compare` и `CompareOrdinal`:

```
public static int Compare (string strA, string strB,
                           StringComparison comparisonType);
public static int Compare (string strA, string strB, bool ignoreCase,
                           CultureInfo culture);
public static int Compare (string strA, string strB, bool ignoreCase);
public static int CompareOrdinal (string strA, string strB);
```

Последние два метода являются просто сокращениями для вызова первых двух методов.

Все методы сравнения порядка возвращают положительное число, отрицательное число или ноль в зависимости от того, как расположено первое значение относительно второго – до, после или рядом:

```
Console.WriteLine ("Boston".CompareTo ("Austin"));       // 1
Console.WriteLine ("Boston".CompareTo ("Boston"));       // 0
Console.WriteLine ("Boston".CompareTo ("Chicago"));      // -1
Console.WriteLine ("Ü".CompareTo ("Ü"));                 // 0
Console.WriteLine ("foo".CompareTo ("FOO"));             // -1
```

В следующем операторе выполняется нечувствительное к регистру сравнение с использованием текущей культуры:

```
Console.WriteLine (string.Compare ("foo", "FOO", true)); // 0
```

Предоставляя объект `CultureInfo`, можно подключить любой алфавит:

```
// Класс CultureInfo определен в пространстве имен System.Globalization
CultureInfo german = CultureInfo.GetCultureInfo ("de-DE");
int i = string.Compare ("Müller", "Muller", false, german);
```

## Класс `StringBuilder`

Класс `StringBuilder` (из пространства имен `System.Text`) представляет изменяемую (редактируемую) строку. С помощью `StringBuilder` можно добавлять (`Append`), вставлять (`Insert`), удалять (`Remove`) и заменять (`Replace`) подстроки, не заменяя целиком объект `StringBuilder`.

Конструктор StringBuilder дополнительно принимает начальное строковое значение, а также стартовый размер для внутренней емкости (по умолчанию 16 символов). Если начальная емкость превышена, то StringBuilder автоматически изменяет размеры своих внутренних структур (за счет небольшого снижения производительности) вплоть до максимальной емкости (которая по умолчанию составляет int.MaxValue).

Популярное применение класса StringBuilder связано с построением длинной строки путем повторяющихся вызовов Append. Такой подход намного более эффективен, чем выполнение множества конкатенаций обычных строковых типов:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 50; i++) sb.Append (i + ",");
```

Для получения финального результата необходимо вызвать ToString():

```
Console.WriteLine (sb.ToString());
```

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
```

В приведенном выше примере выражение i + "," означает, что мы по-прежнему многократно выполняем конкатенацию строк. Тем не менее, такое действие требует лишь незначительных затрат производительности, т.к. строки небольшие и они не растут с каждой новой итерацией цикла. Однако для достижения максимальной производительности тело цикла можно было бы изменить следующим образом:

```
{ sb.Append (i); sb.Append (","); }
```

Метод AppendLine выполняет добавление последовательности новой строки ("\r\n" в Windows). Он принимает смешанную форматную строку в точности как метод String.Format.

Помимо методов Insert, Remove и Replace (метод Replace функционирует подобно методу Replace в типе string) в классе StringBuilder определено свойство Length и записываемый индексатор для получения/установки отдельных символов.

Для очистки содержимого StringBuilder необходимо либо создать новый экземпляр StringBuilder, либо установить его свойство Length в 0.

Установка свойства Length экземпляра StringBuilder в 0 не сокращает его *внутреннюю* емкость. Таким образом, если экземпляр StringBuilder ранее содержал один миллион символов, то после обнуления его свойства Length он продолжит занимать около 2 Мбайт памяти. Чтобы освободить эту память, потребуется создать новый экземпляр StringBuilder и дать возможность старому покинуть область видимости (и подвергнуться сборке мусора).

## Кодировка текста и Unicode

*Набор символов* – это распределение символов, с каждым из которых связан числовой код или *кодовый знак* (code point). Чаще всего используются два набора символов: Unicode и ASCII. Набор Unicode имеет адресное пространство для примерно миллиона символов, из которых в настоящее время распределено около 100 000. Набор Unicode охватывает самые распространенные в мире языки, а также ряд исторических языков и специальных символов. Набор ASCII – это просто первые 128 символов

набора Unicode, которые покрывают большинство из того, что можно видеть на английской клавиатуре. Набор ASCII появился на 30 лет раньше Unicode и продолжает временами применяться из-за своей простоты и эффективности: каждый его символ представлен одним байтом.

Система типов .NET предназначена для работы с набором символов Unicode. Тем не менее, набор ASCII поддерживается неявно в силу того, что является подмножеством Unicode.

*Кодировка текста* отображает числовые кодовые знаки символов на их двоичные представления. В .NET кодировки текста вступают в игру главным образом при работе с текстовыми файлами или потоками. Когда текстовый файл читается с помещением в строку, *кодировщик текста* транслирует данные файла из двоичного представления во внутреннее представление Unicode, которое ожидают типы `char` и `string`. Кодировка текста может ограничивать множество представляемых символов, а также влиять на эффективность хранения.

В .NET имеются две категории кодировок текста:

- кодировки, которые отображают символы Unicode на другой набор символов;
- кодировки, которые используют стандартные схемы кодирования Unicode.

Первая категория содержит унаследованные кодировки, такие как IBM EBCDIC и 8-битные наборы символов с расширенными символами в области из 128 старших символов, которые были популярны до появления Unicode (они идентифицируются кодовой страницей). Кодировка ASCII также относится к данной категории: она кодирует первые 128 символов и отбрасывает остальные. Вдобавок эта категория содержит GB18030 – обязательный стандарт для приложений, которые написаны в Китае (или предназначены для продажи в Китае), начиная с 2000 года.

Во второй категории находятся кодировки UTF-8, UTF-16 и UTF-32 (и устаревшая UTF-7). Каждая из них отличается требованиями к пространству. Кодировка UTF-8 наиболее эффективна с точки зрения пространства для большинства видов текста: при представлении каждого символа в ней применяется *от 1 до 4 байтов*. Первые 128 символов требуют только одного байта, делая эту кодировку совместимой с ASCII. Кодировка UTF-8 чаще всего используется в текстовых файлах и потоках (особенно в Интернете), к тому же она стандартно применяется для потокового ввода-вывода в .NET (фактически она является стандартом почти для всего, что неявно использует кодировку).

Кодировка UTF-16 для представления каждого символа применяет одно или два 16-битных слова и используется внутри .NET для представления символов и строк. Некоторые программы также записывают файлы в UTF-16.

Кодировка UTF-32 наименее экономна в плане пространства: она отображает каждый кодовый знак на 32 бита, т.е. любой символ потребляет 4 байта. По указанной причине кодировка UTF-32 применяется редко. Однако она существенно упрощает произвольный доступ, поскольку каждый символ занимает одинаковое количество байтов.

### Получение объекта `Encoding`

Класс `Encoding` в пространстве имен `System.Text` – это общий базовый класс для классов, инкапсулирующих кодировки текста. Существует несколько подклассов, назначение которых заключается в инкапсуляции семейств кодировок с похожими возможностями. Простейший способ создать экземпляр правильно сконфигурированного класса предусматривает вызов метода `Encoding.GetEncoding` со стандартным

именем набора символов IANA (Internet Assigned Numbers Authority – администрация адресного пространства Интернета):

```
Encoding utf8 = Encoding.GetEncoding ("utf-8");
Encoding chinese = Encoding.GetEncoding ("GB18030");
```

Наиболее распространенные кодировки также могут быть получены через выделенные статические свойства класса `Encoding`:

| Название кодировки | Статическое свойство класса `Encoding` |
|---|---|
| UTF-8 | `Encoding.UTF8` |
| UTF-16 | `Encoding.Unicode` (*не* `UTF16`) |
| UTF-32 | `Encoding.UTF32` |
| ASCII | `Encoding.ASCII` |

Статический метод `GetEncodings` возвращает список всех поддерживаемых кодировок с их стандартными именами IANA:

```
foreach (EncodingInfo info in Encoding.GetEncodings())
  Console.WriteLine (info.Name);
```

Другой способ получения кодировки предполагает создание экземпляра класса кодировки напрямую. В таком случае появляется возможность устанавливать различные варианты через аргументы конструктора, включая описанные ниже.

- Должно ли генерироваться исключение, если при декодировании встречается недопустимая последовательность байтов. По умолчанию исключение не генерируется.
- Должно ли производиться кодирование/декодирование UTF-16/UTF-32 с наиболее значащими байтами вначале (*обратный порядок байтов*) или с наименее значащими байтами вначале (*прямой порядок байтов*). По умолчанию принимается прямой порядок байтов, который является стандартом в операционной системе Windows.
- Должен ли выдаваться маркер порядка байтов (префикс, указывающий конкретный *порядок следования байтов*).

### Кодировка для файлового и потокового ввода-вывода

Наиболее распространенное использование объекта `Encoding` связано с управлением способом чтения и записи в файл или поток. Например, следующий код записывает строку `"testing"` в файл по имени `data.txt` с кодировкой UTF-16:

```
System.IO.File.WriteAllText ("data.txt", "testing", Encoding.Unicode);
```

Если опустить последний аргумент, тогда метод `WriteAllText` применит вездесущую кодировку UTF-8.

UTF-8 является стандартной кодировкой текста для всего файлового и потокового ввода-вывода.

Мы еще вернемся к данной теме в разделе "Адаптеры потоков" главы 15.

### Кодирование и байтовые массивы

Объект Encoding можно также использовать для работы с байтовым массивом. Метод GetBytes преобразует string в byte[] с применением заданной кодировки, а метод GetString преобразует byte[] в string:

```
byte[] utf8Bytes  = System.Text.Encoding.UTF8.GetBytes    ("0123456789");
byte[] utf16Bytes = System.Text.Encoding.Unicode.GetBytes ("0123456789");
byte[] utf32Bytes = System.Text.Encoding.UTF32.GetBytes   ("0123456789");

Console.WriteLine (utf8Bytes.Length);   // 10
Console.WriteLine (utf16Bytes.Length);  // 20
Console.WriteLine (utf32Bytes.Length);  // 40

string original1 = System.Text.Encoding.UTF8.GetString    (utf8Bytes);
string original2 = System.Text.Encoding.Unicode.GetString (utf16Bytes);
string original3 = System.Text.Encoding.UTF32.GetString   (utf32Bytes);

Console.WriteLine (original1);  // 0123456789
Console.WriteLine (original2);  // 0123456789
Console.WriteLine (original3);  // 0123456789
```

### UTF-16 и суррогатные пары

Вспомните, что символы и строки в .NET хранятся в кодировке UTF-16. Поскольку UTF-16 требует одного или двух 16-битных слов на символ, а тип char имеет длину только 16 битов, то некоторые символы Unicode требуют для своего представления два экземпляра char. Отсюда пара следствий:

- свойство Length строки может иметь более высокое значение, чем реальное количество символов;
- одиночного символа char не всегда достаточно для полного представления символа Unicode.

Большинство приложений игнорируют указанные следствия, потому что почти все часто используемые символы попадают внутрь раздела Unicode, который называется *базовой многоязыковой плоскостью* (Basic Multilingual Plane — BMP) и требует только одного 16-битного слова в UTF-16. Плоскость BMP охватывает десятки мировых языков и включает свыше 30 000 китайских иероглифов. Исключениями являются символы ряда древних языков, символы для записи музыкальных произведений и некоторые менее распространенные китайские иероглифы.

Если необходимо поддерживать символы из двух слов, то следующие статические методы в char преобразуют 32-битный кодовый знак в строку из двух char и наоборот:

```
string ConvertFromUtf32 (int utf32)
int    ConvertToUtf32   (char highSurrogate, char lowSurrogate)
```

Символы из двух слов называются *суррогатными*. Их легко обнаружить, поскольку каждое слово находится в диапазоне от 0xD800 до 0xDFFF. Для помощи в этом можно воспользоваться следующими статическими методами в char:

```
bool IsSurrogate     (char c)
bool IsHighSurrogate (char c)
bool IsLowSurrogate  (char c)
bool IsSurrogatePair (char highSurrogate, char lowSurrogate)
```

Класс StringInfo из пространства имен System.Globalization также предлагает ряд методов и свойств для работы с символами из двух слов.

Символы за пределами плоскости BMP обычно требуют специальных шрифтов и имеют ограниченную поддержку в операционных системах.

# Дата и время

Работой по представлению даты и времени занимаются три неизменяемые структуры из пространства имен System: DateTime, DateTimeOffset и TimeSpan. Специальные ключевые слова, которые бы отображались на указанные типы, в языке C# отсутствуют.

## Структура TimeSpan

Структура TimeSpan представляет временной интервал или время суток. В последней роли это просто "часы" (не имеющие даты), которые эквивалентны времени, прошедшему с полуночи, в предположении, что нет перехода на летнее время. Разрешающая способность TimeSpan составляет 100 наносекунд, максимальное значение примерно соответствует 10 миллионам дней, а значение может быть положительным или отрицательным.

Существуют три способа конструирования TimeSpan:

- с помощью одного из конструкторов;
- путем вызова одного из статических методов From...;
- за счет вычитания одного экземпляра DateTime из другого.

Вот доступные конструкторы:

```
public TimeSpan (int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds,
                 int milliseconds);
public TimeSpan (long ticks);  // Каждый тик равен 100 наносекунд
```

Статические методы From... более удобны, когда необходимо указать интервал в каких-то одних единицах, скажем, минутах, часах и т.д.:

```
public static TimeSpan FromDays (double value);
public static TimeSpan FromHours (double value);
public static TimeSpan FromMinutes (double value);
public static TimeSpan FromSeconds (double value);
public static TimeSpan FromMilliseconds (double value);
```

Например:

```
Console.WriteLine (new TimeSpan (2, 30, 0));     // 02:30:00
Console.WriteLine (TimeSpan.FromHours (2.5));    // 02:30:00
Console.WriteLine (TimeSpan.FromHours (-2.5));   // -02:30:00
```

В структуре TimeSpan перегружены операции < и >, а также операции + и -. В результате вычисления приведенного ниже выражения получается значение TimeSpan, соответствующее 2,5 часам:

```
TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30);
```

Следующее выражение дает в результате значение на 1 секунду меньше 10 дней:

```
TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);   // 9.23:59:59
```

С помощью приведенного ниже выражения можно проиллюстрировать работу целочисленных свойств Days, Hours, Minutes, Seconds и Milliseconds:

```
TimeSpan nearlyTenDays = TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);

Console.WriteLine (nearlyTenDays.Days);              // 9
Console.WriteLine (nearlyTenDays.Hours);             // 23
Console.WriteLine (nearlyTenDays.Minutes);           // 59
Console.WriteLine (nearlyTenDays.Seconds);           // 59
Console.WriteLine (nearlyTenDays.Milliseconds);      // 0
```

Напротив, свойства `Total...` возвращают значения типа `double`, описывающие промежуток времени целиком:

```
Console.WriteLine (nearlyTenDays.TotalDays);         // 9.99998842592593
Console.WriteLine (nearlyTenDays.TotalHours);        // 239.999722222222
Console.WriteLine (nearlyTenDays.TotalMinutes);      // 14399.9833333333
Console.WriteLine (nearlyTenDays.TotalSeconds);      // 863999
Console.WriteLine (nearlyTenDays.TotalMilliseconds); // 863999000
```

Статический метод `Parse` представляет собой противоположность `ToString`, преобразуя строку в значение `TimeSpan`. Метод `TryParse` делает то же самое, но в случае неудачного преобразования вместо генерации исключения возвращает `false`. Класс `XmlConvert` также предоставляет методы для преобразования между `TimeSpan` и `string`, которые следуют стандартным протоколам форматирования XML.

Стандартным значением для `TimeSpan` является `TimeSpan.Zero`.

Структуру `TimeSpan` также можно применять для представления времени суток (времени, прошедшего с полуночи). Для получения текущего времени суток необходимо обратиться к свойству `DateTime.Now.TimeOfDay`.

## Структуры DateTime и DateTimeOffset

Типы `DateTime` и `DateTimeOffset` — это неизменяемые структуры для представления даты и дополнительно времени. Их разрешающая способность составляет 100 наносекунд, а поддерживаемый диапазон лет — от 0001 до 9999.

Структура `DateTimeOffset` появилась в .NET Framework 3.5 и функционально похожа на `DateTime`. Ее отличительной особенностью является сохранение также смещения UTC, что позволяет получать более осмысленные результаты при сравнении значений из разных часовых поясов.

В блогах MSDN BCL доступна великолепная статья Энтони Мура с обоснованием введения структуры `DateTimeOffset`, которая называется "A Brief History of `DateTime`" ("Краткая история `DateTime`").

### Выбор между DateTime и DateTimeOffset

Структуры `DateTime` и `DateTimeOffset` отличаются способом обработки часовых поясов. Структура `DateTime` содержит флаг с тремя состояниями, который указывает, относительно чего отсчитывается значение `DateTime`:

- местное время на текущем компьютере;
- UTC (современный эквивалент Гринвичского времени);
- не определено.

Структура `DateTimeOffset` более специфична — она хранит смещение UTC в виде `TimeSpan`:

```
July 01 2018 03:00:00 -06:00
```

Это влияет на сравнения эквивалентности, которые являются главным фактором при выборе между DateTime и DateTimeOffset. В частности:

- во время сравнения DateTime игнорирует флаг с тремя состояниями и считает, что два значения равны, если они имеют одинаковый год, месяц, день, часы, минуты и т.д.;
- структура DateTimeOffset считает два значения равными, если они ссылаются на *ту же самую точку во времени.*

Переход на летнее время может сделать указанные различия важными, даже если приложение не нуждается в учете множества географических часовых поясов.

Таким образом, DateTime считает следующие два значения отличающимися, тогда как DateTimeOffset — равными:

```
July 01 2018 09:00:00 +00:00 (GMT)
July 01 2018 03:00:00 -06:00 (местное время, Центральная Америка)
```

В большинстве случаев логика эквивалентности DateTimeOffset предпочтительнее. Скажем, при выяснении, какое из двух международных событий произошло позже, структура DateTimeOffset даст полностью правильный ответ. Аналогично хакер, планирующий атаку типа распределенного отказа в обслуживании, определенно выберет DateTimeOffset! Чтобы сделать то же самое с помощью DateTime, в приложении потребуется повсеместное приведение к единому часовому поясу (обычно UTC). Такой подход проблематичен по двум причинам.

- Для обеспечения дружественности к конечному пользователю UTC-значения DateTime перед форматированием требуют явного преобразования в местное время.
- Довольно легко забыть и случайно воспользоваться местным значением DateTime.

Однако структура DateTime лучше при указании значения относительно локального компьютера во время выполнения — например, если нужно запланировать архивацию в каждом международном офисе на следующее воскресенье в 3 утра местного времени (когда наблюдается минимальная активность). В такой ситуации больше подойдет структура DateTime, т.к. она будет отражать местное время на каждой площадке.

Внутренне структура DateTimeOffset для хранения смещения UTC в минутах применяет короткий целочисленный тип. Она не хранит никакой региональной информации, так что ничего не указывает на то, к какому времени относится смещение +08:00, например — в Сингапуре или в Перте (Западная Австралия).

Мы рассмотрим часовые пояса и сравнение эквивалентности более подробно в разделе "Даты и часовые пояса" далее в главе.

В SQL Server 2008 была введена прямая поддержка DateTimeOffset через новый тип данных с таким же именем.

### Конструирование DateTime

В структуре DateTime определены конструкторы, которые принимают целочисленные значения для года, месяца и дня, а также дополнительно – для часов, минут, секунд и миллисекунд:

```
public DateTime (int year, int month, int day);
public DateTime (int year, int month, int day,
                 int hour, int minute, int second, int millisecond);
```

Если указывается только дата, то время неявно устанавливается в полночь (0:00).

Конструкторы DateTime также позволяют задавать DateTimeKind – перечисление со следующими значениями:

```
Unspecified, Local, Utc
```

Это соответствует флагу с тремя состояниями, который был описан в предыдущем разделе. Unspecified принимается по умолчанию и означает, что DateTime не зависит от часового пояса. Local означает отношение к местному часовому поясу, установленному на текущем компьютере. Такой экземпляр DateTime не содержит информации о *конкретном часовом поясе* и в отличие от DateTimeOffset не хранит числовое смещение UTC.

Свойство Kind в DateTime возвращает значение DateTimeKind.

Конструкторы DateTime также перегружены с целью приема объекта Calendar, что позволяет указывать дату с использованием подклассов Calendar, которые определены в пространстве имен System.Globalization. Например:

```
DateTime d = new DateTime (5767, 1, 1,
                           new System.Globalization.HebrewCalendar());
Console.WriteLine (d);  // 12/12/2006 12:00:00 AM
```

(Форматирование даты в этом примере зависит от настроек в панели управления на компьютере.) Структура DateTime всегда работает со стандартным григорианским календарем – в приведенном примере во время конструирования происходит однократное преобразование. Для выполнения вычислений с применением другого календаря вы должны применять методы на самом подклассе Calendar.

Конструировать экземпляр DateTime можно также с единственным значением *тиков* типа long, где *тики* представляют собой количество 100-наносекундных интервалов, прошедших с полуночи 01/01/0001.

Для целей взаимодействия DateTime предоставляет статические методы FromFileTime и FromFileTimeUtc, которые обеспечивают преобразование времени файлов Windows (указанного как long), и статический метод FromOADate, преобразующий дату/время автоматизации OLE (типа double).

Чтобы сконструировать экземпляр DateTime из строки, понадобится вызвать статический метод Parse или ParseExact. Оба метода принимают дополнительные флаги и поставщики форматов; ParseExact также принимает форматную строку. Мы обсудим разбор более детально в разделе "Форматирование и разбор" далее в главе.

### Конструирование DateTimeOffset

Структура DateTimeOffset имеет похожий набор конструкторов. Отличие в том, что также указывается смещение UTC в виде TimeSpan:

```
public DateTimeOffset (int year, int month, int day,
                       int hour, int minute, int second,
                       TimeSpan offset);
```

```
public DateTimeOffset (int year, int month, int day,
                       int hour, int minute, int second, int millisecond,
                       TimeSpan offset);
```

Значение TimeSpan должно составлять целое количество минут, иначе сгенерируется исключение.

Вдобавок структура DateTimeOffset имеет конструкторы, которые принимают объект Calendar и значение тиков типа long, а также статические методы Parse и ParseExact, принимающие строку.

Конструировать экземпляр DateTimeOffset можно из существующего экземпляра DateTime либо с использованием перечисленных ниже конструкторов:

```
public DateTimeOffset (DateTime dateTime);
public DateTimeOffset (DateTime dateTime, TimeSpan offset);
```

либо с помощью неявного приведения:

```
DateTimeOffset dt = new DateTime (2000, 2, 3);
```

Неявное приведение DateTime к DateTimeOffset удобно тем, что в большей части .NET Framework поддерживается DateTime, а не DateTimeOffset.

Если смещение не указано, тогда оно выводится из значения DateTime с применением следующих правил:

- если DateTime имеет значение DateTimeKind, равное Utc, то смещение равно нулю;
- если DateTime имеет значение DateTimeKind, равное Local или Unspecified (по умолчанию), то смещение берется из текущего часового пояса.

Для преобразования в другом направлении структура DateTimeOffset предлагает три свойства, которые возвращают значения типа DateTime:

- свойство UtcDateTime возвращает экземпляр DateTime, представленный как время UTC;
- свойство LocalDateTime возвращает экземпляр DateTime в текущем часовом поясе (при необходимости преобразованный);
- свойство DateTime возвращает экземпляр DateTime в любом часовом поясе, который был указан, со свойством Kind, равным Unspecified (т.е. возвращает время UTC плюс смещение).

### Текущие значения DateTime/DateTimeOffset

Обе структуры DateTime и DateTimeOffset имеют статическое свойство Now, которое возвращает текущую дату и время:

```
Console.WriteLine (DateTime.Now);          // 11/11/2017 1:23:45 PM
Console.WriteLine (DateTimeOffset.Now);    // 11/11/2017 1:23:45 PM -06:00
```

Структура DateTime также предоставляет свойство Today, возвращающее порцию даты:

```
Console.WriteLine (DateTime.Today);        // 11/11/2017 12:00:00 AM
```

Статическое свойство UtcNow возвращает текущую дату и время в UTC:

```
Console.WriteLine (DateTime.UtcNow);       // 11/11/2017 7:23:45 AM
Console.WriteLine (DateTimeOffset.UtcNow); // 11/11/2017 7:23:45 AM +00:00
```

Точность всех этих методов зависит от операционной системы и обычно находится в пределах 10–20 миллисекунд.

### Работа с датой и временем

Структуры DateTime и DateTimeOffset предлагают похожий набор свойств экземпляра, которые возвращают элементы даты/времени:

```
DateTime dt = new DateTime (2000, 2, 3,
                            10, 20, 30);

Console.WriteLine (dt.Year);          // 2000
Console.WriteLine (dt.Month);         // 2
Console.WriteLine (dt.Day);           // 3
Console.WriteLine (dt.DayOfWeek);     // Thursday
Console.WriteLine (dt.DayOfYear);     // 34

Console.WriteLine (dt.Hour);          // 10
Console.WriteLine (dt.Minute);        // 20
Console.WriteLine (dt.Second);        // 30
Console.WriteLine (dt.Millisecond);   // 0
Console.WriteLine (dt.Ticks);         // 630851700300000000
Console.WriteLine (dt.TimeOfDay);     // 10:20:30  (возвращает TimeSpan)
```

Структура DateTimeOffset также имеет свойство Offset типа TimeSpan.

Оба типа предоставляют указанные ниже методы экземпляра, которые предназначены для выполнения вычислений (большинство из них принимают аргумент типа double или int):

```
AddYears  AddMonths   AddDays
AddHours  AddMinutes  AddSeconds  AddMilliseconds  AddTicks
```

Все они возвращают новый экземпляр DateTime или DateTimeOffset и учитывают такие аспекты, как високосный год. Для вычитания можно передавать отрицательное значение.

Метод Add добавляет TimeSpan к DateTime или DateTimeOffset. Операция + перегружена для выполнения той же работы:

```
TimeSpan ts = TimeSpan.FromMinutes (90);
Console.WriteLine (dt.Add (ts));
Console.WriteLine (dt + ts);  // то же, что и выше
```

Можно также вычитать TimeSpan из DateTime/DateTimeOffset и вычитать один экземпляр DateTime/DateTimeOffset из другого. Последнее действие дает в результате TimeSpan:

```
DateTime thisYear = new DateTime (2015, 1, 1);
DateTime nextYear = thisYear.AddYears (1);
TimeSpan oneYear = nextYear - thisYear;
```

### Форматирование и разбор даты и времени

Вызов метода ToString на DateTime форматирует результат в виде *краткой даты* (все числа), за которой следует *полное время* (включающее секунды). Например:

```
11/11/2017 11:50:30 AM
```

По умолчанию панель управления операционной системы определяет аспекты вроде того, что должно указываться первым – день, месяц или год, должны ли использоваться ведущие нули и какой формат суток применяется – 12- или 24-часовой.

Вызов метода ToString на DateTimeOffset дает то же самое, но вдобавок возвращает и смещение:

```
11/11/2017 11:50:30 AM -06:00
```

Методы ToShortDateString и ToLongDateString возвращают только часть, касающуюся даты. Формат полной даты также определяется в панели управления; примером может служить "Saturday, 11 November 2017". Методы ToShortTimeString и ToLongTimeString возвращают только часть, касающуюся времени, скажем, 17:10:10 (ToShortTimeString не включает секунды).

Только что описанные четыре метода в действительности являются сокращениями для четырех разных *форматных строк*. Метод ToString перегружен для приема форматной строки и поставщика, позволяя указывать широкий диапазон вариантов и управлять применением региональных настроек. Мы рассмотрим это более подробно в разделе "Форматирование и разбор" далее в главе.

Значения DateTime и DateTimeOffset могут быть некорректно разобраны, если текущие настройки культуры отличаются от настроек, использованных при форматировании. Такой проблемы можно избежать, применяя метод ToString вместе с форматной строкой, которая игнорирует настройки культуры (например, "o"):

```
DateTime dt1 = DateTime.Now;
string cannotBeMisparsed = dt1.ToString ("o");
DateTime dt2 = DateTime.Parse (cannotBeMisparsed);
```

Статические методы Parse/TryParse и ParseExact/TryParseExact выполняют действие, противоположное методу ToString — преобразуют строку в DateTime или DateTimeOffset. Данные методы также имеют перегруженные версии, которые принимают поставщик формата. Вместо генерации исключения FormatException методы Try... возвращают значение false.

### Значения DateTime и DateTimeOffset, равные null

Поскольку DateTime и DateTimeOffset являются структурами, они по своей сути не могут принимать значение null. Когда требуется возможность null, существуют два способа добиться цели:

- использовать тип, допускающий null (т.е. DateTime? или DateTimeOffset?);
- применять статическое поле DateTime.MinValue или DateTimeOffset.MinValue (*стандартные значения* для этих типов).

Использование типов, допускающих null, обычно является более предпочтительным подходом, поскольку в таком случае компилятор помогает предотвращать ошибки. Поле DateTime.MinValue полезно для обеспечения обратной совместимости с кодом, написанным до выхода версии C# 2.0 (в которой появились типы, допускающие null).

Вызов метода ToUniversalTime или ToLocalTime на DateTime.MinValue может дать в результате значение, не являющееся DateTime.MinValue (в зависимости от того, по какую сторону от Гринвича вы находитесь). Если вы находитесь прямо на Гринвиче (Англия в период, когда летнее время не действует), то данная проблема не возникает, потому что местное время и UTC совпадают. Считайте это компенсацией за английскую зиму.

# Даты и часовые пояса

В данном разделе мы более детально исследуем влияние часовых поясов на типы DateTime и DateTimeOffset. Мы также рассмотрим типы TimeZone и TimeZoneInfo, которые предоставляют информацию по смещениям часовых поясов и переходу на летнее время.

## DateTime и часовые пояса

Структура DateTime обрабатывает часовые пояса упрощенным образом. Внутренне DateTime состоит из двух порций информации:

- 62-битное число, которое указывает количество тиков, прошедших с момента 1/1/0001;
- 2-битное значение перечисления DateTimeKind (Unspecified, Local или Utc).

При сравнении двух экземпляров DateTime сравниваются только их значения *тиков*, а значения DateTimeKind игнорируются:

```
DateTime dt1 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Local);
DateTime dt2 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Utc);
Console.WriteLine (dt1 == dt2);   // True
DateTime local = DateTime.Now;
DateTime utc = local.ToUniversalTime();
Console.WriteLine (local == utc);  // False
```

Методы экземпляра ToUniversalTime/ToLocalTime выполняют преобразование в универсальное/местное время. Они применяют текущие настройки часового пояса компьютера и возвращают новый экземпляр DateTime со значением DateTimeKind, равным Utc или Local. При вызове ToUniversalTime на экземпляре DateTime, который уже является Utc, или ToLocalTime на DateTime, который уже представляет собой Local, никакие преобразования не выполняются. Тем не менее, в случае вызова ToUniversalTime или ToLocalTime на экземпляре DateTime, являющемся Unspecified, преобразование произойдет.

С помощью статического метода DateTime.SpecifyKind можно конструировать экземпляр DateTime, который будет отличаться от других только значением поля Kind:

```
DateTime d = new DateTime (2017, 12, 12);    // Unspecified
DateTime utc = DateTime.SpecifyKind (d, DateTimeKind.Utc);
Console.WriteLine (utc);                     // 12/12/2017 12:00:00 AM
```

## DateTimeOffset и часовые пояса

Структура DateTimeOffset внутри содержит поле DateTime, значение которого всегда представлено как UTC, и 16-битное целочисленное поле Offset для смещения UTC, выраженного в минутах. Операции сравнения имеют дело только с полем DateTime (UTC); поле Offset используется главным образом для форматирования.

Методы ToUniversalTime/ToLocalTime возвращают экземпляр DateTimeOffset, представляющий один и тот же момент времени, но в UTC или местном времени. В отличие от DateTime эти методы не воздействуют на лежащее в основе значение даты/времени, а только на смещение:

```
DateTimeOffset local = DateTimeOffset.Now;
DateTimeOffset utc   = local.ToUniversalTime();

Console.WriteLine (local.Offset);      // -06:00:00 (в Центральной Америке)
Console.WriteLine (utc.Offset);        // 00:00:00

Console.WriteLine (local == utc);      // True
```

Чтобы включить в сравнение поле `Offset`, необходимо применить метод `EqualsExact`:

```
Console.WriteLine (local.EqualsExact (utc)); // False
```

## TimeZone и TimeZoneInfo

Классы `TimeZone` и `TimeZoneInfo` предоставляют информацию, касающуюся названий часовых поясов, смещений UTC и правил перехода на летнее время. Класс `TimeZoneInfo` является более мощным; он появился в версии .NET Framework 3.5.

Крупное отличие между указанными двумя типами состоит в том, что `TimeZone` позволяет обращаться только к текущему местному часовому поясу, тогда как `TimeZoneInfo` обеспечивает доступ к часовым поясам по всему миру. Кроме того, `TimeZoneInfo` открывает более обширную (хотя местами и более неуклюжую) модель на основе правил, предназначенную для описания перехода на летнее время.

### Класс TimeZone

Статический метод `TimeZone.CurrentTimeZone` возвращает объект `TimeZone`, основанный на текущих местных настройках. Ниже показаны результаты, которые получены для Калифорнии:

```
TimeZone zone = TimeZone.CurrentTimeZone;
Console.WriteLine (zone.StandardName); // Pacific Standard Time
                                       // (стандартное тихоокеанское время)
Console.WriteLine (zone.DaylightName);  // Pacific Daylight Time
                                        // (летнее тихоокеанское время)
```

Методы `IsDaylightSavingTime` и `GetUtcOffset` работают следующим образом:

```
DateTime dt1 = new DateTime (2018, 1, 1);
DateTime dt2 = new DateTime (2018, 6, 1);
Console.WriteLine (zone.IsDaylightSavingTime (dt1)); // False
Console.WriteLine (zone.IsDaylightSavingTime (dt2)); // True
Console.WriteLine (zone.GetUtcOffset (dt1));         // 08:00:00
Console.WriteLine (zone.GetUtcOffset (dt2));         // 09:00:00
```

Метод `GetDaylightChanges` возвращает специфичную информацию о летнем времени для заданного года:

```
DaylightTime day = zone.GetDaylightChanges (2018);
Console.WriteLine (day.Start.ToString ("M"));        // 11 March
Console.WriteLine (day.End.ToString ("M"));          // 04 November
```

### Класс TimeZoneInfo

Класс `TimeZoneInfo` работает аналогично. Метод `TimeZoneInfo.Local` возвращает текущий местный часовой пояс:

```
TimeZoneInfo zone = TimeZoneInfo.Local;
Console.WriteLine (zone.StandardName); // Pacific Standard Time
                                       // (стандартное тихоокеанское время)
```

```
Console.WriteLine (zone.DaylightName);  // Pacific Daylight Time
                                        // (летнее тихоокеанское время)
```

В классе TimeZoneInfo также доступны методы IsDaylightSavingTime и GetUtcOffset; отличие между ними заключается в том, что один принимает DateTime, а другой — DateTimeOffset.

Вызвав метод FindSystemTimeZoneById с идентификатором пояса, можно получить экземпляр TimeZoneInfo для любого часового пояса в мире. Такая возможность уникальна для TimeZoneInfo, как и все остальные средства, которые будут демонстрироваться, начиная с этого момента. Мы переключимся на Западную Австралию по причинам, которые вскоре станут ясны:

```
TimeZoneInfo wa = TimeZoneInfo.FindSystemTimeZoneById
                ("W. Australia Standard Time");

Console.WriteLine (wa.Id);    // W. Australia Standard Time
                              // (стандартное время Западной Австралии)
Console.WriteLine (wa.DisplayName); // (GMT+08:00) Perth ((GMT+08:00) Перт)
Console.WriteLine (wa.BaseUtcOffset);  // 08:00:00
Console.WriteLine (wa.SupportsDaylightSavingTime);  // True
```

Свойство Id соответствует значению, которое передано методу FindSystemTimeZoneById. Статический метод GetSystemTimeZones возвращает все часовые пояса мира; следовательно, можно вывести список всех допустимых идентификаторов поясов:

```
foreach (TimeZoneInfo z in TimeZoneInfo.GetSystemTimeZones())
  Console.WriteLine (z.Id);
```

Можно также создать специальный часовой пояс, вызвав метод TimeZoneInfo.CreateCustomTimeZone. Поскольку класс TimeZoneInfo неизменяемый, данному методу должны передаваться все существенные данные в качестве аргументов.

С помощью вызова метода ToSerializedString можно сериализировать предопределенный или специальный часовой пояс в (почти) читабельную для человека строку, а посредством вызова метода TimeZoneInfo.FromSerializedString десериализировать ее.

Статический метод ConvertTime преобразует экземпляр DateTime или DateTimeOffset из одного часового пояса в другой. Можно включить либо только целевой объект TimeZoneInfo, либо исходный и целевой объекты TimeZoneInfo. С помощью методов ConvertTimeFromUtc и ConvertTimeToUtc можно также выполнять преобразование прямо из или в UTC.

Для работы с летним временем TimeZoneInfo предоставляет перечисленные ниже дополнительные методы.

- IsInvalidTime возвращает true, если значение DateTime находится в пределах часа (или дельты), который будет пропущен, когда часы переводятся вперед.
- IsAmbiguousTime возвращает true, если DateTime или DateTimeOffset находятся в пределах часа (или дельты), который будет повторен, когда часы переводятся назад.
- GetAmbiguousTimeOffsets возвращает массив из элементов TimeSpan, представляющий допустимые варианты смещения для неоднозначного DateTime или DateTimeOffset.

В отличие от TimeZone из DateZoneInfo нельзя получить простые даты, отражающие начало и конец летнего времени. Взамен должен вызываться метод GetAdjustmentRules, который возвращает декларативный список правил перехода на летнее время, применяемых ко всем годам. Каждое правило имеет DateStart и DateEnd, указывающие диапазон дат, в рамках которого это правило допустимо:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
  Console.WriteLine ("Rule: applies from " + rule.DateStart +
                                     " to " + rule.DateEnd);
```

В Западной Австралии переход на летнее время впервые был введен в 2006 году, в межсезонье (и затем в 2009 году отменен). Это требует специального правила для первого года; таким образом, существуют два правила:

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
Rule: applies from 1/01/2007 12:00:00 AM to 31/12/2009 12:00:00 AM
```

Каждый экземпляр AdjustmentRule имеет свойство DaylightDelta типа TimeSpan (почти в каждом случае оно составляет один час), а также свойства DaylightTransitionStart и DaylightTransitionEnd. Последние два свойства относятся к типу TimeZoneInfo.TransitionTime, в котором определены следующие свойства:

```
public bool IsFixedDateRule { get; }
public DayOfWeek DayOfWeek { get; }
public int Week { get; }
public int Day { get; }
public int Month { get; }
public DateTime TimeOfDay { get; }
```

Время перехода несколько усложняется тем, что должно представлять и фиксированные, и плавающие даты. Примером плавающей даты может служить "последнее воскресенье марта месяца". Ниже описаны правила интерпретации времени перехода.

1. Если для конечного перехода свойство IsFixedDateRule равно true, свойство Day — 1, свойство Month — 1 и свойство TimeOfDay — DateTime.MinValue, то в таком году летнее время не заканчивается (это может произойти только в южном полушарии после первоначального ввода перехода на летнее время в регионе).
2. В противном случае, если IsFixedDateRule равно true, тогда свойства Month, Day и TimeOfDay определяют начало или конец правила корректировки.
3. В противном случае, если IsFixedDateRule равно false, то свойства Month, DayOfWeek, Week и TimeOfDay определяют начало или конец правила корректировки.

В последней ситуации свойство Week ссылается на неделю месяца, при этом 5 означает последнюю неделю. Мы можем проиллюстрировать сказанное путем перечисления правил корректировки для часового пояса wa:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
{
  Console.WriteLine ("Rule: applies from " + rule.DateStart +
                     " to " + rule.DateEnd);  // применяется с ... до ...
  Console.WriteLine ("   Delta: " + rule.DaylightDelta);  // дельта
  Console.WriteLine ("   Start: " + FormatTransitionTime
```

```
                        (rule.DaylightTransitionStart, false));  // начало
  Console.WriteLine ("   End:   " + FormatTransitionTime
                          (rule.DaylightTransitionEnd, true));  // конец
  Console.WriteLine();
}
```

В методе `FormatTransitionTime` мы соблюдаем только что описанные правила:

```
static string FormatTransitionTime (TimeZoneInfo.TransitionTime tt,
                                    bool endTime)
{
  if (endTime && tt.IsFixedDateRule
              && tt.Day == 1 && tt.Month == 1
              && tt.TimeOfDay == DateTime.MinValue)
    return "-";

  string s;
  if (tt.IsFixedDateRule)
    s = tt.Day.ToString();
  else
    s = "The " +
        "first second third fourth last".Split() [tt.Week - 1] +
        " " + tt.DayOfWeek + " in";

  return s + " " + DateTimeFormatInfo.CurrentInfo.MonthNames [tt.Month-1]
         + " at " + tt.TimeOfDay.TimeOfDay;
}
```

Результат для Западной Австралии интересен тем, что он демонстрирует правила и фиксированных, и плавающих дат, а также отсутствие конечной даты:

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
   Delta: 01:00:00
   Start: 3 December at 02:00:00
   End:   -

Rule: applies from 1/01/2007 12:00:00 AM to 31/12/2009 12:00:00 AM
   Delta: 01:00:00
   Start: The last Sunday in October at 02:00:00
   End:   The last Sunday in March at 03:00:00
```

На самом деле Западная Австралия является единственной в таком отношении. Вот как мы обнаружили данный факт:

```
from zone in TimeZoneInfo.GetSystemTimeZones()
let rules = zone.GetAdjustmentRules()
where
  rules.Any
   (r => r.DaylightTransitionEnd.IsFixedDateRule) &&
  rules.Any
   (r => !r.DaylightTransitionEnd.IsFixedDateRule)
select zone
```

## Летнее время и `DateTime`

Если вы используете структуру `DateTimeOffset` или UTC-вариант `DateTime`, то сравнения эквивалентности свободны от влияния летнего времени. Однако с местными вариантами `DateTime` переход на летнее время может вызвать проблемы.

Подытожить правила можно следующим образом.

- Переход на летнее время влияет на местное время, но не на время UTC.
- Когда часы переводят назад, тогда сравнения, основанные на том, что время движется вперед, дадут сбой, если (и только если) они применяют местные значения DateTime.
- Всегда можно надежно перемещаться между UTC и местным временем (на том же самом компьютере), даже когда часы переводят назад.

Метод IsDaylightSavingTime сообщает о том, относится ли заданное местное значение DateTime к летнему времени. В случае времени UTC всегда возвращается false:

```
Console.Write (DateTime.Now.IsDaylightSavingTime());      // True или False
Console.Write (DateTime.UtcNow.IsDaylightSavingTime());  // Всегда False
```

Предполагая, что dto имеет тип DateTimeOffset, следующее выражение делает то же самое:

```
dto.LocalDateTime.IsDaylightSavingTime
```

Конец летнего времени представляет отдельную сложность для алгоритмов, использующих местное время. Когда часы переводят назад, то один и тот же час (точнее Delta) повторяется. Мы можем продемонстрировать это, создав экземпляр DateTime прямо в "пограничной зоне" на компьютере и затем вычтя Delta:

```
DaylightTime changes = TimeZone.CurrentTimeZone.GetDaylightChanges (2010);
TimeSpan halfDelta = new TimeSpan (changes.Delta.Ticks / 2);
DateTime utc1 = changes.End.ToUniversalTime() - halfDelta;
DateTime utc2 = utc1 - changes.Delta;
```

Преобразование переменных utc1 и utc2 в местное время показывает, почему следует применять UTC, а не местное время, если код полагается на то, что время движется вперед:

```
DateTime loc1 = utc1.ToLocalTime();     // (стандартное тихоокеанское время)
DateTime loc2 = utc2.ToLocalTime();
Console.WriteLine (loc1);               // 2/11/2010 1:30:00 AM
Console.WriteLine (loc2);               // 2/11/2010 1:30:00 AM
Console.WriteLine (loc1 == loc2);       // True
```

Несмотря на сообщение о том, что значения переменных loc1 и loc2 равны, внутри они разные. Тип DateTime резервирует специальный бит для указания, с какой стороны пограничной зоны расположена неоднозначная дата. При сравнении данный бит игнорируется, как было показано выше, но вступает в игру при форматировании DateTime:

```
Console.Write (loc1.ToString ("o")); // 2010-11-02T02:30:00.0000000-08:00
Console.Write (loc2.ToString ("o")); // 2010-11-02T02:30:00.0000000-07:00
```

Этот бит также читается при преобразовании обратно в UTC, обеспечивая успешные перемещения между местным временем и UTC:

```
Console.WriteLine (loc1.ToUniversalTime() == utc1); // True
Console.WriteLine (loc2.ToUniversalTime() == utc2); // True
```

Любые два экземпляра DateTime можно надежно сравнивать, предварительно вызывая на каждом метод ToUniversalTime. Такая стратегия отказывает, если (и только если) в точности один из них имеет значение DateTimeKind, равное Unspecified. Возможность отказа является еще одной причиной отдавать предпочтение типу DateTimeOffset.

# Форматирование и разбор

Форматирование означает преобразование *в* строку, а разбор – преобразование *из* строки. Потребность в форматировании и разборе во время программирования возникает часто, причем в самых разнообразных ситуациях. Для этого в .NET Framework предусмотрено несколько механизмов.

- **`ToString` и `Parse`.** Данные методы предоставляют стандартную функциональность для многих типов.
- **Поставщики форматов.** Они проявляются в виде дополнительных методов `ToString` (и `Parse`), которые принимают форматную строку и/или поставщик формата. Поставщики форматов характеризуются высокой гибкостью и чувствительностью к культуре. В состав .NET Framework входят поставщики форматов для числовых типов и типов `DateTime`/`DateTimeOffset`.
- **`XmlConvert`.** Это статический класс с методами, которые поддерживают форматирование и разбор с соблюдением стандартов XML. Класс `XmlConvert` также удобен при универсальном преобразовании, когда требуется обеспечить независимость от культуры либо избежать некорректного разбора. Класс `XmlConvert` поддерживает числовые типы, а также типы `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan` и `Guid`.
- **Преобразователи типов.** Они предназначены для визуальных конструкторов и средств разбора XAML.

В настоящем разделе мы обсудим первые два механизма, уделяя особое внимание поставщикам форматов. В следующем разделе мы опишем `XmlConvert` и преобразователи типов, а также другие механизмы преобразования.

## `ToString` и `Parse`

Метод `ToString` является простейшим механизмом форматирования. Он обеспечивает осмысленный вывод для всех простых типов значений (т.е. `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan`, `Guid` и всех числовых типов). Для обратной операции в каждом из указанных типов определен статический метод `Parse`. Например:

```
string s = true.ToString();   // s = "True"
bool b = bool.Parse (s);      // b = true
```

Если разбор терпит неудачу, то генерируется исключение `FormatException`. Во многих типах также определен метод `TryParse`, который в случае отказа преобразования вместо генерации исключения возвращает `false`:

```
int i;
bool failure = int.TryParse ("qwerty", out i);
bool success = int.TryParse ("123", out i);
```

Если вы ожидаете ошибку, тогда вызов `TryParse` будет более быстрым и элегантным решением, чем вызов `Parse` в блоке обработки исключения.

Методы `Parse` и `TryParse` в `DateTime` (`DateTimeOffset`) и числовых типах учитывают местные настройки культуры; это можно изменить, указывая объект `CultureInfo`. Часто указание инвариантной культуры является удачной идеей. Например, разбор `"1.234"` в `double` дает `1234` для Германии:

```
Console.WriteLine (double.Parse ("1.234")); // 1234 (в Германии)
```

Причина в том, что символ точки в Германии используется в качестве разделителя тысяч, а не как десятичная точка. Указание инвариантной культуры исправляет ситуацию:

```
double x = double.Parse ("1.234", CultureInfo.InvariantCulture);
```

То же самое применимо и в отношении вызова ToString:

```
string x = 1.234.ToString (CultureInfo.InvariantCulture);
```

## Поставщики форматов

Временами требуется больший контроль над тем, как происходит форматирование и разбор. Например, существуют десятки способов форматирования DateTime (DateTimeOffset). Поставщики форматов позволяют получить обширный контроль над форматированием и разбором и поддерживаются для числовых типов и типов даты/времени. Поставщики форматов также используются элементами управления пользовательского интерфейса для выполнения форматирования и разбора.

Интерфейсом для применения поставщика формата является IFormattable, который реализуют все числовые типы и тип DateTime (DateTimeOffset):

```
public interface IFormattable
{
  string ToString (string format, IFormatProvider formatProvider);
}
```

Методу ToString в первом аргументе передается *форматная строка*, а во втором – *поставщик формата*. Форматная строка предоставляет инструкции; поставщик формата определяет то, как инструкции транслируются. Например:

```
NumberFormatInfo f = new NumberFormatInfo();
f.CurrencySymbol = "$$";
Console.WriteLine (3.ToString ("C", f));  // $$ 3.00
```

Здесь "C" представляет собой форматную строку, которая указывает *денежное значение*, а объект NumberFormatInfo является поставщиком формата, определяющим то, каким образом должно визуализироваться денежное значение (и другие числовые представления). Такой механизм допускает глобализацию.

Все форматные строки для чисел и дат описаны в разделе "Стандартные форматные строки и флаги разбора" далее в главе.

Если для форматной строки или поставщика указать null, то будет применен стандартный вариант. Стандартный поставщик формата – CultureInfo.CurrentCulture, который, если только он не переустановлен, отражает настройки панели управления компьютера во время выполнения. Например:

```
Console.WriteLine (10.3.ToString ("C", null));  // $10.30
```

Для удобства в большинстве типов метод ToString перегружен, так что null для поставщика можно не указывать:

```
Console.WriteLine (10.3.ToString ("C")); // $10.30
Console.WriteLine (10.3.ToString ("F4")); //10.3000 (четыре десятичных позиции)
```

Вызов метода ToString без аргументов для типа DateTime (DateTimeOffset) или числового типа эквивалентен использованию стандартного поставщика формата с пустой форматной строкой.

В .NET Framework определены три поставщика формата (все они реализуют интерфейс IFormatProvider):

```
NumberFormatInfo
DateTimeFormatInfo
CultureInfo
```

Все типы перечислений также поддерживают форматирование, хотя специальный класс, реализующий интерфейс IFormatProvider, для них не предусмотрен.

### Поставщики форматов и CultureInfo

В рамках контекста поставщиков форматов тип CultureInfo действует как механизм косвенности для двух других поставщиков форматов, возвращая объект NumberFormatInfo или DateTimeFormatInfo, который может быть применен к региональным настройкам культуры.

В следующем примере мы запрашиваем специфическую культуру (английский (*en*glish) в Великобритании (*G*reat *B*ritain)):

```
CultureInfo uk = CultureInfo.GetCultureInfo ("en-GB");
Console.WriteLine (3.ToString ("C", uk));  // £3.00
```

Показанный код выполняется с использованием стандартного объекта NumberFormatInfo, применимого к культуре en-GB.

В приведенном ниже примере производится форматирование DateTime с использованием инвариантной культуры. Инвариантная культура всегда остается одной и той же независимо от настроек компьютера:

```
DateTime dt = new DateTime (2000, 1, 2);
CultureInfo iv = CultureInfo.InvariantCulture;
Console.WriteLine (dt.ToString (iv));           // 01/02/2000 00:00:00
Console.WriteLine (dt.ToString ("d", iv));      // 01/02/2000
```

Инвариантная культура основана на американской культуре с перечисленными ниже отличиями:

- символом валюты является ¤, а не $;
- дата и время форматируются с ведущими нулями (хотя месяц по-прежнему идет первым);
- для времени применяется 24-часовой формат, а не 12-часовой с указателем AM/PM.

### Использование NumberFormatInfo или DateTimeFormatInfo

В следующем примере мы создаем экземпляр NumberFormatInfo и изменяем разделитель групп цифр с запятой на пробел. После этого мы применяем его для форматирования числа с тремя десятичными позициями:

```
NumberFormatInfo f = new NumberFormatInfo ();
f.NumberGroupSeparator = " ";
Console.WriteLine (12345.6789.ToString ("N3", f));  // 12 345.679
```

Начальные настройки для NumberFormatInfo или DateTimeFormatInfo основаны на инвариантной культуре. Тем не менее, иногда более удобно выбирать другую стартовую точку. Для этого можно клонировать с помощью Clone существующий поставщик формата:

```
NumberFormatInfo f = (NumberFormatInfo)
                     CultureInfo.CurrentCulture.NumberFormat.Clone();
```

Клонированный поставщик формата всегда является записываемым, даже если исходный поставщик допускал только чтение.

### Смешанное форматирование

Смешанные форматные строки позволяют комбинировать подстановку переменных с форматными строками. Статический метод `string.Format` принимает смешанную форматную строку (мы иллюстрировали это в разделе "Обработка строк и текста" в начале главы):

```
string composite = "Credit={0:C}";
Console.WriteLine (string.Format (composite, 500));  // Credit=$500.00
```

Класс `Console` перегружает свои методы `Write` и `WriteLine` для приема смешанных форматных строк, позволяя несколько сократить код примера:

```
Console.WriteLine ("Credit={0:C}", 500);             // Credit=$500.00
```

Смешанную форматную строку можно также добавлять к `StringBuilder` (через `AppendFormat`) и к `TextWriter` для ввода-вывода (глава 15).

Метод `string.Format` принимает необязательный поставщик формата. Простым его применением является вызов `ToString` на произвольном объекте с передачей в то же время поставщика формата. Например:

```
string s = string.Format (CultureInfo.InvariantCulture, "{0}", someObject);
```

Код эквивалентен следующему коду:

```
string s;
if (someObject is IFormattable)
  s = ((IFormattable)someObject).ToString (null, CultureInfo.InvariantCulture);
else if (someObject == null)
  s = "";
else
  s = someObject.ToString();
```

### Разбор с использованием поставщиков форматов

Стандартного интерфейса для выполнения разбора посредством поставщика формата не предусмотрено. Взамен каждый участвующий тип имеет перегруженную версию своего статического метода `Parse` (и `TryParse`), которая принимает поставщик формата и дополнительно значение перечисления `NumberStyles` или `DateTimeStyles`.

Перечисления `NumberStyles` и `DateTimeStyles` управляют работой разбора: они позволяют указывать аспекты наподобие того, могут ли встречаться во входной строке круглые скобки или символ валюты. (По умолчанию ни то, ни другое *не* разрешено.) Например:

```
int error = int.Parse ("(2)");  // Генерируется исключение

int minusTwo = int.Parse ("(2)", NumberStyles.Integer |
                                 NumberStyles.AllowParentheses);  // Нормально

decimal fivePointTwo = decimal.Parse ("£5.20", NumberStyles.Currency,
                                      CultureInfo.GetCultureInfo ("en-GB"));
```

В следующем разделе описаны все члены перечислений `NumberStyles` и `DateTimeStyles`, а также стандартные правила разбора для каждого типа.

### IFormatProvider и ICustomFormatter

Все поставщики форматов реализуют интерфейс IFormatProvider:

```
public interface IFormatProvider { object GetFormat (Type formatType); }
```

Цель заключается в обеспечении косвенности — именно это позволяет CultureInfo поручить выполнение работы соответствующему объекту NumberFormatInfo или DateTimeInfo.

За счет реализации интерфейса IFormatProvider — наряду с ICustomFormatter — можно также построить собственный поставщик формата в сочетании с существующими типами. В интерфейсе ICustomFormatter определен единственный метод:

```
string Format (string format, object arg, IFormatProvider formatProvider);
```

Следующий поставщик формата записывает числа с помощью слов:

```
// Программа доступна для загрузки по адресу http://www.albahari.com/nutshell/
public class WordyFormatProvider : IFormatProvider, ICustomFormatter
{
  static readonly string[] _numberWords =
   "zero one two three four five six seven eight nine minus point".Split();

  IFormatProvider _parent;  // Позволяет потребителям строить цепочки
                            // поставщиков форматов

  public WordyFormatProvider () : this (CultureInfo.CurrentCulture) { }
  public WordyFormatProvider (IFormatProvider parent)
  {
    _parent = parent;
  }

  public object GetFormat (Type formatType)
  {
    if (formatType == typeof (ICustomFormatter)) return this;
    return null;
  }

  public string Format (string format, object arg, IFormatProvider prov)
  {
    // Если это не наша форматная строка,
    // тогда передать ее родительскому поставщику:
    if (arg == null || format != "W")
      return string.Format (_parent, "{0:" + format + "}", arg);

    StringBuilder result = new StringBuilder();
    string digitList = string.Format (CultureInfo.InvariantCulture,
                                      "{0}", arg);
    foreach (char digit in digitList)
    {
      int i = "0123456789-.".IndexOf (digit);
      if (i == -1) continue;
      if (result.Length > 0) result.Append (' ');
      result.Append (_numberWords[i]);
    }
    return result.ToString();
  }
}
```

Обратите внимание, что в методе Format мы применяем string.Format для преобразования входного числа в строку с указанием InvariantCulture. Было бы

намного проще вызвать ToString на arg, но тогда задействовалось бы свойство CurrentCulture. Причина потребности в инвариантной культуре становится очевидной дальше в коде:

```
int i = "0123456789-.".IndexOf (digit);
```

Здесь критически важно, чтобы строка с числом содержала только символы 0123456789-. и никаких интернационализированных версий для них.

Ниже приведен пример использования WordyFormatProvider:

```
double n = -123.45;
IFormatProvider fp = new WordyFormatProvider();
Console.WriteLine (string.Format (fp, "{0:C} in words is {0:W}", n));

// Выводит -$123.45 in words is minus one two three point four five
```

Специальные поставщики форматов могут применяться только в смешанных форматных строках.

# Стандартные форматные строки и флаги разбора

Стандартные форматные строки управляют способом преобразования в строку числового типа или типа DateTime/DateTimeOffset. Существуют два вида форматных строк.

- **Стандартные форматные строки.** С их помощью обеспечивается общее управление. Стандартная форматная строка состоит из одиночной буквы и следующей за ней дополнительной цифры (смысл которой зависит от буквы). Примером может служить "C" или "F2".
- **Специальные форматные строки.** С их помощью контролируется каждый символ посредством шаблона. Примером может служить "0:#.000E+00".

Специальные форматные строки не имеют никакого отношения к специальным поставщикам форматов.

## Форматные строки для чисел

В табл. 6.2 приведен список всех стандартных форматных строк для чисел.

**Таблица 6.2. Стандартные форматные строки для чисел**

| Буква | Что означает | Пример ввода | Результат | Примечания |
|---|---|---|---|---|
| G или g | "Общий" формат | 1.2345, "G"<br>0.00001, "G"<br>0.00001, "g"<br>1.2345, "G3"<br>12345, "G3" | 1.2345<br>1E-05<br>1e-05<br>1.23<br>1.23E04 | Переключается на экспоненциальную запись для очень малых или больших чисел. G3 ограничивает точность *всего* тремя цифрами (перед и после точки) |
| F | Формат с фиксированной точкой | 2345.678, "F2"<br>2345.6, "F2" | 2345.68<br>2345.60 | F2 округляет до двух десятичных позиций |

| Буква | Что означает | Пример ввода | Результат | Примечания |
|---|---|---|---|---|
| N | Формат с фиксированной точкой и *разделителем групп* ("числовой") | 2345.678, "N2"<br>2345.6, "N2" | 2,345.68<br>2,345.60 | То же, что и выше, но с разделителем групп (тысяч); детали берутся из поставщика формата |
| D | Заполнение ведущими нулями | 123, "D5"<br>123, "D1" | 00123<br>123 | Только для целочисленных типов.<br>D5 дополняет слева до пяти цифр; усечение не производится |
| E или e | Принудительно применение экспоненциальной записи | 56789, "E"<br>56789, "e"<br>56789, "E2" | 5.678900E+004<br>5.678900e+004<br>5.68E+004 | По умолчанию точность составляет шесть цифр |
| C | Денежное значение | 1.2, "C"<br>1.2, "C4" | $1.20<br>$1.2000 | C без цифры использует стандартное количество десятичных позиций, заданное поставщиком формата |
| P | Процент | .503, "P"<br>.503, "P0" | 50.30 %<br>50 % | Использует символ и компоновку из поставщика формата<br>Десятичные позиции могут быть отброшены |
| X или x | Шестнадцатеричный формат | 47, "X"<br>47, "x"<br>47, "X4" | 2F<br>2f<br>002F | X — для представления шестнадцатеричных цифр в верхнем регистре; x — для представления шестнадцатеричных цифр в нижнем регистре. Только для целочисленных типов |
| R | Округление | 1f / 3f, "R" | 0.333333343 | Для типов float и double с помощью форматной строки R или G17 производится округление |

Предоставление форматной строки не для чисел (либо null или пустой строки) эквивалентно применению стандартной форматной строки "G" без цифры. В этом случае поведение будет следующим.

- Числа меньше $10^{-4}$ или больше, чем точность типа, выражаются с использованием экспоненциальной (научной) записи.
- Две десятичные позиции на пределе точности float или double округляются, чтобы замаскировать неточности, присущие преобразованию в десятичный тип из лежащей в основе двоичной формы.

Только что описанное автоматическое округление обычно полезно и проходит незаметно. Тем не менее, оно может вызвать проблему, если необходимо вернуться обратно к числу; другими словами, преобразование числа в строку и обратно (возможно, многократно повторяемое) может нарушить равенство значений. По этой причине существуют форматные строки "R" и "G17", подавляющие такое неявное округление.

Начиная с .NET Framework 4.6, форматные строки "R" и "G17" выполняют одно и то же действие; в предшествующих версиях .NET Frameworks форматная строка "R" по существу является ошибочной версией "G17" и применяться не должна.

В табл. 6.3 представлен список специальных форматных строк для чисел.

**Таблица 6.3. Специальные форматные строки для чисел**

| Спецификатор | Что означает | Пример ввода | Результат | Примечания |
|---|---|---|---|---|
| # | Заполнитель для цифры | 12.345, ".##"<br>12.345, ".####" | 12.35<br>12.345 | Ограничивает количество цифр после десятичной точки |
| 0 | Заполнитель для нуля | 12.345, ".00"<br>12.345, ".0000"<br>99, "000.00" | 12.35<br>12.3450<br>099.00 | Как и выше, ограничивает количество цифр после десятичной точки, но также дополняет нулями до и после десятичных позиций |
| . | Десятичная точка | | | Отображает десятичную точку.<br>Действительный символ берется из NumberFormatInfo |
| , | Разделитель групп | 1234, "#,###,###"<br>1234, "0,000,000" | 1,234<br>0,001,234 | Символ берется из NumberFormatInfo |
| , (как и выше) | Коэффициент | 1000000, "#,"<br>1000000, "#,," | 1000<br>1 | Когда запятая находится до или после десятичной позиции, она действует как коэффициент, разделяя результат на 1000, 1 000 000 и т.д. |
| % | Процентная запись | 0.6, "00%" | 60% | Сначала умножает на 100, а затем подставляет символ процента, полученный из NumberFormatInfo |
| E0, e0, E+0, e+0, E-0, e-0 | Экспоненциальная запись | 1234, "0E0"<br>1234, "0E+0"<br>1234, "0.00E00"<br>1234, "0.00e00" | 1E3<br>1E+3<br>1.23E03<br>1.23e03 | |

| Спецификатор | Что означает | Пример ввода | Результат | Примечания |
|---|---|---|---|---|
| \ | Признак литерального символа | `50, @"\#0"` | `#50` | Используется в сочетании с префиксом @ в строках — или же можно применять \\ |
| 'xx' | Признак литеральной строки | `50, "0 '...'"` | `50 ...` | |
| ; | Разделитель секций | `15, "#;(#);zero"`<br>`-5, "#;(#);zero"`<br>`0, "#;(#);zero"` | `15`<br>`(5)`<br>`zero` | (Если положительное)<br>(Если отрицательное)<br>(Если ноль) |
| Любой другой символ | Литерал | `35.2, "$0 . 00c"` | `$35 . 20c` | |

## Перечисление NumberStyles

В каждом числовом типе определен статический метод `Parse`, принимающий аргумент типа `NumberStyles`. Перечисление флагов `NumberStyles` позволяет определить, каким образом строка читается при преобразовании в числовой тип. Перечисление `NumberStyles` имеет следующие комбинируемые члены:

```
AllowLeadingWhite        AllowTrailingWhite
AllowLeadingSign         AllowTrailingSign
AllowParentheses         AllowDecimalPoint
AllowThousands           AllowExponent
AllowCurrencySymbol      AllowHexSpecifier
```

В `NumberStyles` также определены составные члены:

```
None  Integer  Float  Number  HexNumber  Currency  Any
```

Все составные члены кроме `None` включают `AllowLeadingWhite` и `AllowTrailingWhite`. Остальные члены проиллюстрированы на рис. 6.1; три наиболее полезных выделены полужирным.

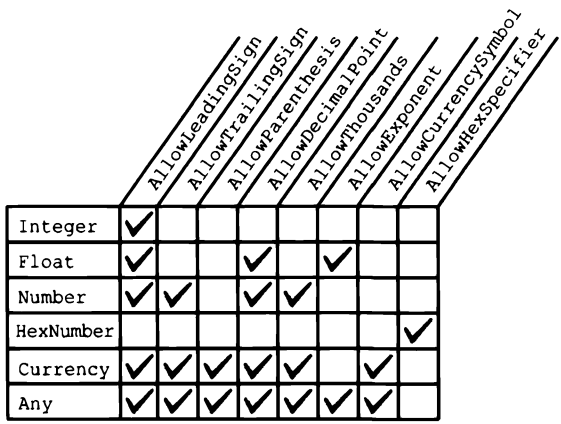



*Рис. 6.1. Составные члены `NumberStyles`*

Когда метод Parse вызывается без указания флагов, применяются правила по умолчанию, как показано на рис. 6.2.

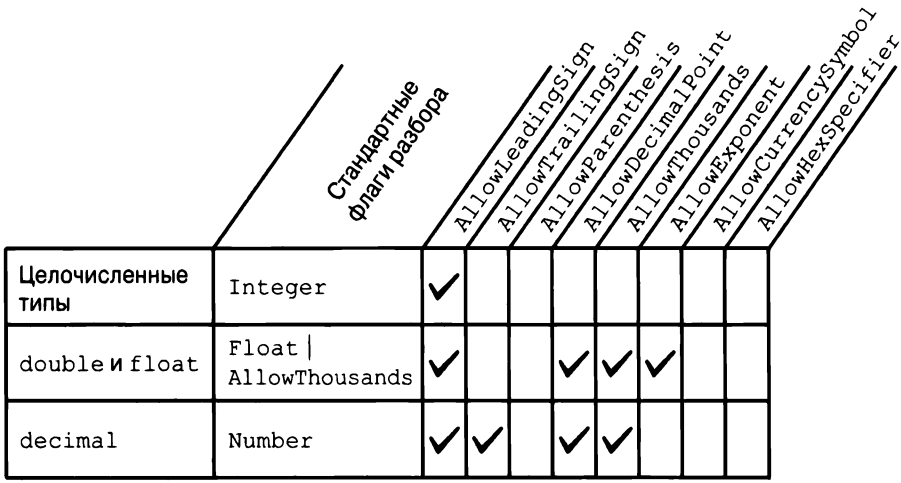



***Рис. 6.2.** Стандартные флаги разбора для числовых типов*

Если правила по умолчанию, представленные на рис. 6.2, не подходят, тогда значения NumberStyles должны указываться явно:

```
int thousand = int.Parse ("3E8", NumberStyles.HexNumber);
int minusTwo = int.Parse ("(2)", NumberStyles.Integer |
                                 NumberStyles.AllowParentheses);
double aMillion = double.Parse ("1,000,000", NumberStyles.Any);
decimal threeMillion = decimal.Parse ("3e6", NumberStyles.Any);
decimal fivePointTwo = decimal.Parse ("$5.20", NumberStyles.Currency);
```

Из-за того, что поставщик формата не указан, приведенный выше код работает с местным символом валюты, разделителем групп, десятичной точкой и т.д. В следующем примере для денежных значений жестко закодирован знак евро и разделитель групп в виде пробела:

```
NumberFormatInfo ni = new NumberFormatInfo();
ni.CurrencySymbol = "€";
ni.CurrencyGroupSeparator = " ";
double million = double.Parse ("€1 000 000", NumberStyles.Currency, ni);
```

## Форматные строки для даты/времени

Форматные строки для DateTime/DateTimeOffset могут быть разделены на две группы на основе того, учитывают ли они настройки культуры и поставщика формата. Форматные строки для даты/времени, чувствительные к культуре, описаны в табл. 6.4, а те, что не чувствительны — в табл. 6.5. Пример вывода получен в результате форматирования следующего экземпляра DateTime (с *инвариантной культурой* в случае табл. 6.4):

```
new DateTime (2000, 1, 2,  17, 18, 19);
```

**Таблица 6.4. Форматные строки для даты/времени, чувствительные к культуре**

| Форматная строка | Что означает | Пример вывода |
|---|---|---|
| d | Краткая дата | 01/02/2000 |
| D | Полная дата | Sunday, 02 January 2000 |
| t | Краткое время | 17:18 |
| T | Полное время | 17:18:19 |
| f | Полная дата + краткое время | Sunday, 02 January 2000 17:18 |
| F | Полная дата + полное время | Sunday, 02 January 2000 17:18:19 |
| g | Краткая дата + краткое время | 01/02/2000 17:18 |
| G (по умолчанию) | Краткая дата + полное время | 01/02/2000 17:18:19 |
| m, M | Месяц и день | 02 January |
| y, Y | Год и месяц | January 2000 |

**Таблица 6.5. Форматные строки для даты/времени, нечувствительные к культуре**

| Форматная строка | Что означает | Пример вывода | Примечания |
|---|---|---|---|
| o | Возможность кругового преобразования | 2000-01-02T 17:18:19.0000000 | Будет присоединять информацию о часовом поясе, если только DateTimeKind не является Unspecified |
| r, R | Стандарт RFC 1123 | Sun, 02 Jan 2000 17:18:19 GMT | Потребуется явно преобразовать в UTC с помощью DateTime.ToUniversalTime |
| s | Сортируемое; ISO 8601 | 2000-01-02T17:18:19 | Совместимо с текстовой сортировкой |
| u | "Универсальное" сортируемое | 2000-01-02 17:18:19Z | Подобно предыдущему; потребуется явно преобразовать в UTC |
| U | UTC | Sunday, 02 January 2000 17:18:19 | Краткая дата + краткое время, преобразованное в UTC |

Форматные строки "r", "R" и "u" выдают суффикс, который подразумевает UTC; пока что они не осуществляют автоматическое преобразование местной версии DateTime в UTC-версию (поэтому преобразование придется делать самостоятельно). По иронии судьбы "U" автоматически преобразует в UTC, но не записывает суффикс часового пояса! На самом деле "o" является единственным спецификатором формата в группе, который обеспечивает запись недвусмысленного экземпляра DateTime безо всякого вмешательства.

Класс DateTimeFormatInfo также поддерживает специальные форматные строки: они аналогичны специальным форматным строкам для чисел. Их полный список можно найти в MSDN. Ниже приведен пример специальной форматной строки:

```
yyyy-MM-dd HH:mm:ss
```

### Разбор и некорректный разбор значений DateTime

Строки, в которых месяц или день помещен первым, являются неоднозначными и очень легко могут привести к некорректному разбору, в частности, если пользователь проживает за пределами США. Это не является проблемой в элементах управления пользовательского интерфейса, т.к. при разборе и форматировании принудительно применяются одни и те же настройки. Но при записи в файл, например, некорректный разбор дня/месяца может стать реальной проблемой. Существуют два решения:

- при форматировании и разборе всегда придерживаться одной и той же явной культуры (например, инвариантной);
- форматировать DateTime и DateTimeOffset в *независимой* от культуры манере.

Второй подход более надежен – особенно если выбран формат, в котором первым указывается год из четырех цифр: такие строки намного реже некорректно разбираются другой стороной. Кроме того, строки, сформатированные с использованием *соответствующего стандартам* формата с годом вначале (таким как "o"), могут корректно разбираться вместе с локально сформатированными строками. (Даты, сформатированные посредством "s" или "u", обеспечивают дополнительное преимущество, будучи сортируемыми.)

В целях иллюстрации предположим, что сгенерирована следующая нечувствительная к культуре строка DateTime по имени s:

```
string s = DateTime.Now.ToString ("o");
```

Форматная строка "o" включает в вывод миллисекунды. Приведенная ниже специальная форматная строка дает тот же результат, что и "o", но без миллисекунд:

```
yyyy-MM-ddTHH:mm:ss K
```

Повторно разобрать строку s можно двумя путями. Метод ParseExact требует строгого соответствия с указанной форматной строкой:

```
DateTime dt1 = DateTime.ParseExact (s, "o", null);
```

(Достичь похожего результата можно с помощью методов ToString и ToDateTime класса XmlConvert.)

Тем не менее, метод Parse неявно принимает как формат "o", так и формат CurrentCulture:

```
DateTime dt2 = DateTime.Parse (s);
```

Прием работает и для DateTime, и для DateTimeOffset.

Метод ParseExact обычно предпочтительнее, если вы знаете формат разбираемой строки. Это означает, что если строка сформатирована некорректно, тогда сгенерируется исключение – что обычно лучше, чем риск получения неправильно разобранной даты.

## Перечисление DateTimeStyles

Перечисление флагов DateTimeStyles предоставляет дополнительные инструкции при вызове метода Parse на DateTime (DateTimeOffset). Ниже приведены его члены:

```
None,
AllowLeadingWhite, AllowTrailingWhite, AllowInnerWhite,
AssumeLocal, AssumeUniversal, AdjustToUniversal,
NoCurrentDateDefault, RoundTripKind
```

Имеется также составной член `AllowWhiteSpaces`:

```
AllowWhiteSpaces = AllowLeadingWhite | AllowTrailingWhite | AllowInnerWhite
```

Стандартным значением является `None`. Таким образом, лишние пробельные символы обычно запрещены (к пробельным символам, являющимся частью стандартного шаблона `DateTime`, это не относится).

Флаги `AssumeLocal` и `AssumeUniversal` применяются, если строка не имеет суффикса часового пояса (такого как `Z` или `+9:00`). Флаг `AdjustToUniversal` учитывает суффиксы часовых поясов, но затем выполняет преобразование в UTC с использованием текущих региональных настроек.

При разборе строки, содержащей время и не включающей дату, по умолчанию берется сегодняшняя дата. Однако если применен флаг `NoCurrentDateDefault`, то будет использоваться 1 января 0001 года.

## Форматные строки для перечислений

В разделе "Перечисления" главы 3 мы описали форматирование и разбор перечислимых значений. В табл. 6.6 показан список форматных строк для перечислений и результаты их применения в следующем операторе:

```
Console.WriteLine (System.ConsoleColor.Red.ToString (formatString));
```

**Таблица 6.6. Форматные строки для перечислений**

| Форматная строка | Что означает | Пример вывода | Примечания |
|---|---|---|---|
| `G` или `g` | "Общий" формат | `Red` | Используется по умолчанию |
| `F` или `f` | Трактуется, как если бы присутствовал атрибут `Flags` | `Red` | Работает на составных членах, даже если перечисление не имеет атрибута `Flags` |
| `D` или `d` | Десятичное значение | `12` | Извлекает лежащее в основе целочисленное значение |
| `X` или `x` | Шестнадцатеричное значение | `0000000C` | Извлекает лежащее в основе целочисленное значение |

# Другие механизмы преобразования

В предшествующих двух разделах рассматривались поставщики форматов – основной механизм .NET для форматирования и разбора. Другие важные механизмы преобразования разбросаны по различным типам и пространствам имен. Некоторые из них преобразуют в и из типа `string`, а некоторые осуществляют другие виды преобразований. В настоящем разделе мы обсудим следующие темы.

- Класс `Convert` и его функции:
  - преобразования вещественных чисел в целые, которые производят округление, а не усечение;

- разбор чисел в системах счисления с основаниями 2, 8 и 16;
- динамические преобразования;
- преобразования Base 64.

- Класс `XmlConvert` и его роль в форматировании и разборе для XML.
- Преобразователи типов и их роль в форматировании и разборе для визуальных конструкторов и XAML.
- Класс `BitConverter`, предназначенный для двоичных преобразований.

## Класс `Convert`

Следующие типы в .NET Framework называются *базовыми типами*:

- `bool`, `char`, `string`, `System.DateTime` и `System.DateTimeOffset`;
- все числовые типы C#.

В статическом классе `Convert` определены методы для преобразования каждого базового типа в любой другой базовый тип. К сожалению, большинство таких методов бесполезны: они либо генерируют исключения, либо избыточны из-за доступности неявных приведений. Тем не менее, среди этого беспорядка есть несколько полезных методов, которые рассматриваются в последующих разделах.

Все базовые типы (явно) реализуют интерфейс `IConvertible`, в котором определены методы для преобразования во все другие базовые типы. В большинстве случаев реализация каждого из этих методов просто вызывает какой-то метод из класса `Convert`. В редких ситуациях может оказаться удобным написание метода, принимающего аргумент типа `IConvertible`.

### Округляющие преобразования вещественных чисел в целые

В главе 2 было показано, что неявные и явные приведения позволяют выполнять преобразования между числовыми типами. Подведем итоги:

- неявные приведения работают для преобразований без потери (например, `int` в `double`);
- явные приведения обязательны для преобразований с потерей (например, `double` в `int`).

Приведения оптимизированы для обеспечения эффективности, поэтому они *усекают* данные, которые не умещаются. В результате может возникнуть проблема при преобразовании вещественного числа в целое, т.к. часто требуется не усечение, а *округление*. Упомянутую проблему решают методы числового преобразования `Convert`; они всегда производят *округление*:

```
double d = 3.9;
int i = Convert.ToInt32 (d);  // i == 4
```

Класс `Convert` использует округление, принятое в банках, при котором серединные значения привязываются к четным целым (это позволяет избегать положительного или отрицательного отклонения). Если округление, принятое в банках, становится проблемой, тогда для вещественного числа необходимо вызвать метод `Math.Round`: он принимает дополнительный аргумент, позволяющий управлять округлением серединного значения.

### Разбор чисел в системах счисления с основаниями 2, 8 и 16

Среди методов To*Целочисленный-тип* скрываются перегруженные версии, которые разбирают числа в системах счисления с другими основаниями:

```
int thirty = Convert.ToInt32  ("1E", 16);    // Разобрать в шестнадцатеричной
                                             // системе счисления
uint five  = Convert.ToUInt32 ("101", 2);    // Разобрать в двоичной системе
                                             // счисления
```

Во втором аргументе задается основание системы счисления. Допускается указывать 2, 8, 10 или 16.

### Динамические преобразования

Иногда требуется осуществлять преобразование из одного типа в другой, но точные типы не известны вплоть до времени выполнения. Для таких целей класс `Convert` предлагает метод `ChangeType`:

```
public static object ChangeType (object value, Type conversionType);
```

Исходный и целевой типы должны относиться к "базовым" типам. Метод `ChangeType` также принимает необязательный аргумент `IFormatProvider`. Ниже представлен пример:

```
Type targetType = typeof (int);
object source = "42";

object result = Convert.ChangeType (source, targetType);

Console.WriteLine (result);            // 42
Console.WriteLine (result.GetType());  // System.Int32
```

Примером, когда это может оказаться полезным, является написание десериализатора, который способен работать с множеством типов. Он также может преобразовывать любое перечисление в его целочисленный тип (как будет показано в разделе "Перечисления" далее в главе).

Ограничение метода `ChangeType` состоит в том, что для него нельзя указывать форматную строку или флаг разбора.

### Преобразования Base 64

Временами возникает необходимость включать двоичные данные вроде растрового изображения в текстовый документ, такой как XML-файл или сообщение электронной почты. Base 64 — повсеместно применяемое средство кодирования двоичных данных в виде читабельных символов, которое использует 64 символа из набора ASCII.

Метод `ToBase64String` класса `Convert` осуществляет преобразование байтового массива в код Base 64, а метод `FromBase64String` выполняет обратное преобразование.

## Класс `XmlConvert`

Класс `XmlConvert` (из пространства имен `System.Xml`) предлагает наиболее подходящие методы для форматирования и разбора данных, которые поступают из XML-файла или направляются в такой файл. Методы в `XmlConvert` учитывают нюансы XML-форматирования, не требуя указания специальных форматных строк. Например, значение `true` в XML выглядит как `true`, но не `True`. Класс `XmlConvert` широко применяется в .NET Framework. Он также хорошо подходит для универсальной и не зависящей от культуры сериализации.

Все методы форматирования в XmlConvert доступны в виде перегруженных версий методов ToString; методы разбора называются ToBoolean, ToDateTime и т.д. Например:

```
string s = XmlConvert.ToString (true); // s = "true"
bool isTrue = XmlConvert.ToBoolean (s);
```

Методы, выполняющие преобразование в и из DateTime, принимают аргумент типа XmlDateTimeSerializationMode – перечисление со следующими значениями:

```
Unspecified, Local, Utc, RoundtripKind
```

Значения Local и Utc вызывают преобразование во время форматирования (если DateTime еще не находится в нужном часовом поясе). Часовой пояс затем добавляется к строке:

```
2010-02-22T14:08:30.9375              // Unspecified
2010-02-22T14:07:30.9375+09:00        // Local
2010-02-22T05:08:30.9375Z             // Utc
```

Значение Unspecified приводит к отбрасыванию перед форматированием любой информации о часовом поясе, встроенной в DateTime (т.е. DateTimeKind). Значение RoundtripKind учитывает DateTimeKind из DateTime, так что при восстановлении результирующая структура DateTime будет в точности такой же, какой была изначально.

## Преобразователи типов

Преобразователи типов предназначены для форматирования и разбора в средах, используемых во время проектирования. Преобразователи типов вдобавок поддерживают разбор значений в документах XAML (Extensible Application Markup Language – расширяемый язык разметки приложений), которые применяются в инфраструктурах Windows Presentation Foundation и Workflow Foundation.

В .NET Framework существует свыше 100 преобразователей типов, покрывающих такие аспекты, как цвета, изображения и URI. В отличие от них поставщики форматов реализованы только для небольшого количества простых типов значений.

Преобразователи типов обычно разбирают строки разнообразными путями, не требуя подсказок. Например, если в приложении ASP.NET внутри Visual Studio присвоить свойству BackColor элемента управления значение, введя **"Beige"** в окне свойств, то преобразователь типа Color определит, что производится ссылка на имя цвета, а не на строку RGB или системный цвет. Временами подобная гибкость может делать преобразователи типов удобными в контекстах, выходящих за рамки визуальных конструкторов и XAML-документов.

Все преобразователи типов являются подклассами класса TypeConverter из пространства имен System.ComponentModel. Для получения экземпляра TypeConverter необходимо вызвать метод TypeDescriptor.GetConverter. Следующий код получает экземпляр TypeConverter для типа Color (из пространства имен System.Drawing в сборке *System.Drawing.dll*):

```
TypeConverter cc = TypeDescriptor.GetConverter (typeof (Color));
```

Среди многих других методов в классе TypeConverter определены методы ConvertToString и ConvertFromString. Их можно вызывать так, как показано ниже:

```
Color beige  = (Color) cc.ConvertFromString ("Beige");
Color purple = (Color) cc.ConvertFromString ("#800080");
Color window = (Color) cc.ConvertFromString ("Window");
```

По соглашению преобразователи типов имеют имена, заканчивающиеся на *Converter*, и обычно находятся в том же самом пространстве имен, что и тип, для которого они предназначены. Тип ссылается на своего преобразователя через атрибут `TypeConverterAttribute`, позволяя визуальным конструкторам автоматически выбирать преобразователи.

Преобразователи типов могут также предоставлять службы времени проектирования, такие как генерация списков стандартных значений для заполнения раскрывающихся списков в визуальном конструкторе или для помощи в написании кода сериализации.

### Класс `BitConverter`

Большинство базовых типов может быть преобразовано в байтовый массив путем вызова метода `BitConverter.GetBytes`:

```
foreach (byte b in BitConverter.GetBytes (3.5))
  Console.Write (b + " ");  // 0 0 0 0 0 0 12 64
```

Класс `BitConverter` также предоставляет методы для преобразования в другом направлении, например, `ToDouble`.

Типы `decimal` и `DateTime` (`DateTimeOffset`) не поддерживаются классом `BitConverter`. Однако значение `decimal` можно преобразовать в массив `int`, вызвав метод `decimal.GetBits`. Для обратного направления тип `decimal` предлагает конструктор, принимающий массив `int`.

В случае типа `DateTime` можно вызывать метод `ToBinary` на экземпляре — в результате возвращается значение `long` (на котором затем можно использовать `BitConverter`). Статический метод `DateTime.FromBinary` выполняет обратное действие.

## Глобализация

С *интернационализацией* приложения связаны два аспекта: *глобализация* и *локализация*.

*Глобализация* имеет отношение к следующим трем задачам (указанным в порядке убывания важности).

1. Обеспечение *работоспособности* программы при запуске на машине с другой культурой.
2. Соблюдение правил форматирования локальной культуры — например, при отображении дат.
3. Проектирование программы таким образом, чтобы она выбирала специфичные к культуре данные и строки из подчиненных сборок, которые можно написать и развернуть позже.

*Локализация* означает написание подчиненных сборок для специфических культур. Это может делаться *после* написания программы — мы раскроем соответствующие детали в разделе "Ресурсы и подчиненные сборки" главы 18.

Платформа .NET Framework содействует решению второй задачи, применяя специфичные для культуры правила по умолчанию. Мы уже показывали, что вызов метода `ToString` на `DateTime` или числе учитывает локальные правила форматирования. К сожалению, в таком случае очень легко допустить ошибку при решении первой задачи и нарушить работу программы, поскольку ожидается, что даты или числа должны быть сформатированы в соответствии с предполагаемой культурой.

Как уже было показано, решение предусматривает либо указание культуры (такой как инвариантная культура) при форматировании и разборе, либо использование независимых от культуры методов вроде тех, которые определены в классе `XmlConvert`.

## Контрольный перечень глобализации

Мы уже рассмотрели в этой главе важные моменты, связанные с глобализацией. Ниже приведен сводный перечень основных требований.

- Освойте Unicode и текстовые кодировки (см. раздел "Кодировка текста и Unicode" ранее в главе).
- Помните о том, что такие методы, как `ToUpper` и `ToLower` в типах `char` и `string`, чувствительны к культуре: если чувствительность к культуре не нужна, тогда применяйте методы `ToUpperInvariant/ToLowerInvariant`.
- Отдавайте предпочтение независимым от культуры механизмам форматирования и разбора для `DateTime` и `DateTimeOffset`, таким как `ToString("o")` и `XmlConvert`.
- В других обстоятельствах указывайте культуру при форматировании/разборе чисел или даты/времени (если только вас не *интересует* поведение локальной культуры).

## Тестирование

Проводить тестирование для различных культур можно путем переустановки свойства `CurrentCulture` класса `Thread` (из пространства имен `System.Threading`). В представленном далее коде текущая культура изменяется на турецкую:

```
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo ("tr-TR");
```

Турецкая культура является очень хорошим тестовым сценарием по следующим причинам.

- `"i".ToUpper() != "I"` и `"I".ToLower() != "i"`.
- Даты форматируются как день.месяц.год (обратите внимание на разделитель в виде точки).
- Символом десятичной точки является запятая, а не собственно точка.

Можно также поэкспериментировать, изменяя настройки форматирования чисел и дат в панели управления Windows: они отражены в стандартной культуре (`CultureInfo.CurrentCulture`).

Метод `CultureInfo.GetCultures` возвращает массив всех доступных культур.

Классы `Thread` и `CultureInfo` также поддерживают свойство `CurrentUICulture`. Оно больше связано с локализацией: мы рассмотрим его в главе 18.

# Работа с числами

## Преобразования

Числовые преобразования были описаны в предшествующих главах и разделах; все доступные варианты подытожены в табл. 6.7.

**Таблица 6.7. Обзор числовых преобразований**

| Задача | Функции | Примеры |
|---|---|---|
| Разбор десятичных чисел | `Parse`<br>`TryParse` | `double d =`<br>`  double.Parse ("3.5");`<br>`int i;`<br>`bool ok =`<br>`  int.TryParse ("3", out i);` |
| Разбор чисел в системах счисления с основаниями 2, 8 или 16 | `Convert.To`*Целочисленный-тип* | `int i =`<br>`Convert.ToInt32("1E", 16);` |
| Форматирование в шестнадцатеричную запись | `ToString ("X")` | `string hex =`<br>`  45.ToString ("X");` |
| Числовое преобразование без потерь | Неявное приведение | `int i = 23;`<br>`double d = i;` |
| Числовое преобразование с *усечением* | Явное приведение | `double d = 23.5;`<br>`int i = (int) d;` |
| Числовое преобразование с *округлением* (вещественных чисел в целые) | `Convert.To`*Целочисленный-тип* | `double d = 23.5;`<br>`int i =`<br>`  Convert.ToInt32 (d);` |

## Класс `Math`

В табл. 6.8 приведен список членов статического класса `Math`. Тригонометрические функции принимают аргументы типа `double`; другие методы наподобие `Max` перегружены для работы со всеми числовыми типами. В классе `Math` также определены математические константы `E` ($e$) и `PI` ($\pi$).

Метод `Round` позволяет указывать количество десятичных позиций для округления, а также способ обработки серединных значений (от нуля или посредством округления, принятого в банках). Методы `Floor` и `Ceiling` округляют до ближайшего целого: `Floor` всегда округляет в меньшую сторону, а `Ceiling` — в бо́льшую, даже отрицательные числа.

Методы `Max` и `Min` принимают только два аргумента. Для массива или последовательности чисел следует применять расширяющие методы `Max` и `Min` из класса `System.Linq.Enumerable`.

**Таблица 6.8. Методы статического класса `Math`**

| Категория | Методы |
|---|---|
| Округление | `Round`, `Truncate`, `Floor`, `Ceiling` |
| Максимум и минимум | `Max`, `Min` |
| Абсолютное значение и знак | `Abs`, `Sign` |
| Квадратный корень | `Sqrt` |
| Возведение в степень | `Pow`, `Exp` |
| Логарифм | `Log`, `Log10` |
| Тригонометрические функции | `Sin`, `Cos`, `Tan`<br>`Sinh`, `Cosh`, `Tanh`<br>`Asin`, `Acos`, `Atan` |

## Структура `BigInteger`

Структура `BigInteger` — это специализированный числовой тип, появившийся в версии .NET Framework 4.0. Он находится в новом пространстве имен `System.Numerics` внутри сборки *`System.Numerics.dll`* и позволяет представлять произвольно большие целые числа без потери точности.

В языке C# отсутствует собственная поддержка `BigInteger`, так что нет никакого способа записи литералов типа `BigInteger`. Тем не менее, можно выполнять неявное преобразование в `BigInteger` из любого другого целочисленного типа. Например:

```
BigInteger twentyFive = 25; // Неявное преобразование из целочисленного типа
```

Чтобы представить крупное число вроде одного гугола ($10^{100}$), можно воспользоваться одним из статических методов `BigInteger`, например, `Pow` (возведение в степень):

```
BigInteger googol = BigInteger.Pow (10, 100);
```

Или же можно применить метод `Parse` для разбора строки:

```
BigInteger googol = BigInteger.Parse ("1".PadRight (100, '0'));
```

Вызов метода `ToString` в данном случае приводит к выводу на экран всех цифр:

```
Console.WriteLine (googol.ToString()); // 10000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
```

Между `BigInteger` и стандартными числовыми типами можно выполнять преобразования с потенциальной потерей точности, используя явную операцию приведения:

```
double g2 = (double) googol;       // Явное приведение
BigInteger g3 = (BigInteger) g2;   // Явное приведение
Console.WriteLine (g3);
```

Вывод демонстрирует потерю точности:

```
99999999999999996733616880411669l2...
```

В структуре `BigInteger` перегружены все арифметические операции, включая получение остатка от деления (`%`), а также операции сравнения и эквивалентности.

Структуру BigInteger можно конструировать из байтового массива. Следующий код генерирует 32-байтовое случайное число, подходящее для криптографии, и затем присваивает его переменной BigInteger:

```
// Здесь используется пространство имен System.Security.Cryptography:
RandomNumberGenerator rand = RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);
var bigRandomNumber = new BigInteger (bytes);  // Преобразовать в BigInteger
```

Преимущество хранения таких чисел в структуре BigInteger по сравнению с байтовым массивом связано с тем, что вы получаете семантику типа значения. Вызов ToByteArray осуществляет преобразование BigInteger обратно в байтовый массив.

## Структура Complex

Структура Complex является еще одним специализированным числовым типом, появившимся в версии .NET Framework 4.0, и предназначена для представления комплексных чисел с помощью вещественных и мнимых компонент типа double. Структура Complex находится в сборке *System.Numerics.dll* (вместе с BigInteger).

Для применения Complex необходимо создать экземпляр структуры, указав вещественное и мнимое значения:

```
var c1 = new Complex (2, 3.5);
var c2 = new Complex (3, 0);
```

Существуют также неявные преобразования в Complex из стандартных числовых типов.

Структура Complex предлагает свойства для вещественного и мнимого значений, а также для фазы и амплитуды:

```
Console.WriteLine (c1.Real);          // 2
Console.WriteLine (c1.Imaginary);     // 3.5
Console.WriteLine (c1.Phase);         // 1.05165021254837
Console.WriteLine (c1.Magnitude);     // 4.03112887414927
```

Конструировать число Complex можно путем указания амплитуды и фазы:

```
Complex c3 = Complex.FromPolarCoordinates (1.3, 5);
```

Стандартные арифметические операции перегружены для работы с числами Complex:

```
Console.WriteLine (c1 + c2);  // (5, 3.5)
Console.WriteLine (c1 * c2);  // (6, 10.5)
```

Структура Complex предоставляет статические методы для выполнения более сложных функций, включая:

- тригонометрические (Sin, Asin, Sinh, Tan и т.д.);
- логарифмы и возведение в степень;
- сопряженное число (Conjugate).

## Класс Random

Класс Random генерирует псевдослучайную последовательность случайных значений byte, int или double.

Чтобы использовать класс Random, сначала надо создать его экземпляр, дополнительно предоставив начальное значение для инициирования последовательности случайных чисел. Применение одного и того же начального значения гарантирует получение той же самой последовательности чисел (при запуске под управлением одной и той же версии CLR), что иногда полезно, когда нужна воспроизводимость:

```
Random r1 = new Random (1);
Random r2 = new Random (1);
Console.WriteLine (r1.Next (100) + ", " + r1.Next (100));  // 24, 11
Console.WriteLine (r2.Next (100) + ", " + r2.Next (100));  // 24, 11
```

Если воспроизводимость не требуется, тогда можно конструировать Random без начального значения — в качестве такого значения будет использоваться текущее системное время.

Поскольку системные часы имеют ограниченный квант времени, два экземпляра Random, созданные в близкие друг к другу моменты времени (обычно в пределах 10 миллисекунд), будут выдавать одинаковые последовательности значений. В общем случае эту проблему решают путем создания нового объекта Random каждый раз, когда требуется случайное число, вместо повторного использования *того же самого* объекта.

Удачный прием предусматривает объявление единственного статического экземпляра Random. Однако в многопоточных сценариях это может привести к проблемам, т.к. объекты Random не являются безопасными в отношении потоков. В разделе "Локальное хранилище потока" главы 22 мы опишем обходной способ.

Вызов метода Next(*n*) генерирует случайное целочисленное значение между 0 и *n*-1. Вызов метода NextDouble генерирует случайное значение double между 0 и 1. Вызов метода NextBytes заполняет случайными значениями байтовый массив.

Класс Random не считается в достаточной степени случайным для приложений, предъявляющих высокие требования к безопасности, к которым относятся, например, криптографические приложения. Для этого .NET Framework предлагает *криптографически строгий* генератор случайных чисел в пространстве имен System.Security.Cryptography. Вот как он применяется:

```
var rand = System.Security.Cryptography.RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);  // Заполнить байтовый массив случайными числами
```

Недостаток данного генератора в том, что он менее гибкий: заполнение байтового массива является единственным средством получения случайных чисел. Чтобы получить целое число, потребуется использовать BitConverter:

```
byte[] bytes = new byte [4];
rand.GetBytes (bytes);
int i = BitConverter.ToInt32 (bytes, 0);
```

# Перечисления

В главе 3 мы описали тип enum и показали, как комбинировать его члены, проверять эквивалентность, применять логические операции и выполнять преобразования. Поддержка перечислений C# в .NET Framework расширяется через тип System.Enum, который выступает в двух ролях:

- обеспечивает унификацию для всех типов enum;
- определяет статические служебные методы.

*Унификация типов* означает возможность неявного приведения любого члена перечисления к экземпляру System.Enum:

```
enum Nut  { Walnut, Hazelnut, Macadamia }
enum Size { Small, Medium, Large }

static void Main()
{
  Display (Nut.Macadamia);     // Nut.Macadamia
  Display (Size.Large);        // Size.Large
}

static void Display (Enum value)
{
  Console.WriteLine (value.GetType().Name + "." + value.ToString());
}
```

Статические служебные методы System.Enum относятся главным образом к выполнению преобразований и получению списков членов.

## Преобразования для перечислений

Представить значение перечисления можно тремя путями:

- как член enum;
- как лежащее в основе целочисленное значение;
- как строку.

В этом разделе мы опишем, как осуществлять преобразования между всеми представлениями.

### Преобразования экземпляра enum в целое значение

Вспомните, что явное приведение выполняет преобразование между членом enum и его целочисленным значением. Явное приведение будет корректным подходом, если тип enum известен на этапе компиляции:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
int i = (int) BorderSides.Top;          // i == 4
BorderSides side = (BorderSides) i;     // side == BorderSides.Top
```

Таким же способом можно приводить экземпляр System.Enum к его целочисленному типу. Трюк заключается в приведении сначала к object, а затем к целочисленному типу:

```
static int GetIntegralValue (Enum anyEnum)
{
  return (int) (object) anyEnum;
}
```

Код полагается на то, что вам известен целочисленный тип: приведенный выше метод потерпит неудачу, если ему передать значение enum, целочисленным типом которого является long. Чтобы написать метод, работающий с enum любого целочисленного типа, можно воспользоваться одним из трех подходов. Первый из них предусматривает вызов метода Convert.ToDecimal:

```
static decimal GetAnyIntegralValue (Enum anyEnum)
{
  return Convert.ToDecimal (anyEnum);
}
```

Данный подход работает, потому что каждый целочисленный тип (включая `ulong`) может быть преобразован в десятичный тип без потери информации. Второй подход предполагает вызов метода `Enum.GetUnderlyingType` для получения целочисленного типа `enum` и затем вызов метода `Convert.ChangeType`:

```
static object GetBoxedIntegralValue (Enum anyEnum)
{
  Type integralType = Enum.GetUnderlyingType (anyEnum.GetType());
  return Convert.ChangeType (anyEnum, integralType);
}
```

Как показано в следующем примере, в результате предохраняется исходный целочисленный тип:

```
object result = GetBoxedIntegralValue (BorderSides.Top);
Console.WriteLine (result);            // 4
Console.WriteLine (result.GetType())   // System.Int32
```

Наш метод `GetBoxedIntegralType` фактически не выполняет никаких преобразований значения; взамен он *переупаковывает* то же самое значение в другой тип. Он транслирует целочисленное значение внутри оболочки *типа перечисления* в целое значение внутри оболочки *целочисленного типа*. Мы рассмотрим это более подробно в разделе "Как работают перечисления" далее в главе.

Третий подход заключается в вызове метода `Format` или `ToString` с указанием форматной строки `"d"` или `"D"`. В итоге получается целочисленное значение `enum` в виде строки, что удобно при написании специальных форматеров сериализации:

```
static string GetIntegralValueAsString (Enum anyEnum)
{
  return anyEnum.ToString ("D");  // Возвращает что-то наподобие "4"
}
```

### Преобразования целочисленного значения в экземпляр `enum`

Метод `Enum.ToObject` преобразует целочисленное значение в экземпляр `enum` заданного типа:

```
object bs = Enum.ToObject (typeof (BorderSides), 3);
Console.WriteLine (bs);  // Left, Right
```

Это динамический эквивалент следующего кода:

```
BorderSides bs = (BorderSides) 3;
```

Метод `ToObject` перегружен для приема всех целочисленных типов, а также типа `object`. (Последняя перегруженная версия работает с любым упакованным целочисленным типом.)

### Строковые преобразования

Для преобразования `enum` в строку можно вызвать либо статический метод `Enum.Format`, либо метод `ToString` на экземпляре. Оба метода принимают формат-

ную строку, которой может быть "G" для стандартного поведения форматирования, "D" для выдачи лежащего в основе целочисленного значения в виде строки, "X" для выдачи лежащего в основе целочисленного значения в виде шестнадцатеричной записи или "F" для форматирования комбинированных членов перечисления без атрибута Flags. Примеры приводились в разделе "Стандартные форматные строки и флаги разбора" ранее в главе.

Метод Enum.Parse преобразует строку в enum. Он принимает тип enum и строку, которая может содержать множество членов:

```
BorderSides leftRight = (BorderSides) Enum.Parse (typeof (BorderSides),
                                                  "Left, Right");
```

Необязательный третий аргумент позволяет выполнять разбор, нечувствительный к регистру. Если член не найден, тогда генерируется исключение ArgumentException.

## Перечисление значений enum

Метод Enum.GetValues возвращает массив, содержащий все члены указанного типа enum:

```
foreach (Enum value in Enum.GetValues (typeof (BorderSides)))
  Console.WriteLine (value);
```

В массив включаются и составные члены, такие как LeftRight = Left | Right.

Метод Enum.GetNames выполняет то же самое действие, но возвращает массив *строк*.

Внутренне среда CLR реализует методы GetValues и GetNames, выполняя рефлексию полей в типе enum. В целях эффективности результаты кешируются.

## Как работают перечисления

Семантика типов enum в значительной степени поддерживается компилятором. В среде CLR во время выполнения не делается никаких отличий между экземпляром enum (когда он не упакован) и лежащим в его основе целочисленным значением. Более того, определение enum в CLR является просто подтипом System.Enum со статическими полями целочисленного типа для каждого члена. В итоге обычное применение enum становится высокоэффективным, приводя к затратам во время выполнения, которые соответствуют затратам, связанным с целочисленными константами.

Недостаток данной стратегии состоит в том, что типы enum могут обеспечивать *статическую*, но не *строгую* безопасность типов. Мы приводили пример в главе 3:

```
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
BorderSides b = BorderSides.Left;
b += 1234;  // Ошибка не возникает!
```

Когда компилятор не имеет возможности выполнить проверку достоверности (как в показанном примере), то нет никакой подстраховки со стороны исполняющей среды в форме генерации исключения.

Может показаться, что утверждение об отсутствии разницы между экземпляром enum и его целочисленным значением на этапе выполнения вступает в противоречие со следующим кодом:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
Console.WriteLine (BorderSides.Right.ToString());        // Right
Console.WriteLine (BorderSides.Right.GetType().Name);    // BorderSides
```

Учитывая природу экземпляра `enum` во время выполнения, можно было бы ожидать вывода на экран `2` и `Int32`! Причина такого поведения кроется в определенных уловках, предпринимаемых компилятором. Компилятор C# явно *упаковывает* экземпляр `enum` перед вызовом его виртуальных методов, таких как `ToString` и `GetType`. И когда экземпляр `enum` упакован, он получает оболочку времени выполнения, которая ссылается на его тип `enum`.

# Структура `Guid`

Структура `Guid` представляет глобально уникальный идентификатор: 16-байтовое значение, которое после генерации является почти наверняка уникальным в мире. Идентификаторы `Guid` часто используются для ключей различных видов — в приложениях и базах данных. Количество уникальных идентификаторов `Guid` составляет $2^{128}$, или $3{,}4\times10^{38}$.

Статический метод `Guid.NewGuid` генерирует уникальный идентификатор `Guid`:

```
Guid g = Guid.NewGuid ();
Console.WriteLine (g.ToString());  // 0d57629c-7d6e-4847-97cb-9e2fc25083fe
```

Для создания идентификатора `Guid` с применением существующего значения предназначено несколько конструкторов. Вот два наиболее полезных из них:

```
public Guid (byte[] b);  // Принимает 16-байтовый массив
public Guid (string g);  // Принимает форматированную строку
```

В случае представления в виде строки идентификатор `Guid` форматируется как шестнадцатеричное число из 32 цифр с необязательными символами - после 8-й, 12-й, 16-й и 20-й цифры. Вся строка также может быть дополнительно помещена в квадратные или фигурные скобки:

```
Guid g1 = new Guid ("{0d57629c-7d6e-4847-97cb-9e2fc25083fe}");
Guid g2 = new Guid ("0d57629c7d6e484797cb9e2fc25083fe");
Console.WriteLine (g1 == g2);  // True
```

Будучи структурой, `Guid` поддерживает семантику типа значения; следовательно, в показанном выше примере операция эквивалентности работает нормально.

Метод `ToByteArray` преобразует `Guid` в байтовый массив.

Статическое свойство `Guid.Empty` возвращает пустой идентификатор `Guid` (со всеми нулями). Оно часто используется вместо `null`.

# Сравнение эквивалентности

До сих пор мы предполагали, что все применяемые операции == и != выполняли сравнение эквивалентности. Однако проблема эквивалентности является более сложной и тонкой, временами требуя использования дополнительных методов и интерфейсов. В настоящем разделе мы исследуем стандартные протоколы C# и .NET для определения эквивалентности, уделяя особое внимание двум вопросам.

- Когда операции == и != адекватны (либо неадекватны) для сравнения эквивалентности, и какие существуют альтернативы?
- Как и когда должна настраиваться логика эквивалентности типа?

Но перед тем как погрузиться в исследование протоколов эквивалентности и способов их настройки мы должны взглянуть на вводные концепции эквивалентности значений и ссылочной эквивалентности.

## Эквивалентность значений и ссылочная эквивалентность

Различают два вида эквивалентности.

- **Эквивалентность значений.** Два значения *эквивалентны* в некотором смысле.
- **Ссылочная эквивалентность.** Две ссылки ссылаются *в точности на один и тот же объект.*

По умолчанию:

- типы значений применяют *эквивалентность значений;*
- ссылочные типы используют *ссылочную эквивалентность.*

На самом деле типы значений могут применять *только* эквивалентность значений (если они не упакованы). Простой демонстрацией эквивалентности значений может служить сравнение двух чисел:

```
int x = 5, y = 5;
Console.WriteLine (x == y);        // True (в силу эквивалентности значений)
```

Более сложная демонстрация предусматривает сравнение двух структур `DateTimeOffset`. Приведенный ниже код выводит на экран `True`, т.к. две структуры `DateTimeOffset` относятся к *одной и той же точке во времени*, а потому считаются эквивалентными:

```
var dt1 = new DateTimeOffset (2010, 1, 1, 1, 1, 1, TimeSpan.FromHours(8));
var dt2 = new DateTimeOffset (2010, 1, 1, 2, 1, 1, TimeSpan.FromHours(9));
Console.WriteLine (dt1 == dt2);    // True
```

Тип `DateTimeOffset` — это структура, семантика эквивалентности которой была подстроена. По умолчанию структуры поддерживают специальный вид эквивалентности значений, называемый *структурной эквивалентностью*, при которой два значения считаются эквивалентными, если эквивалентны все их члены. (Вы можете удостовериться в сказанном, создав структуру и вызвав ее метод `Equals`; позже будут приведены более подробные обсуждения.)

Ссылочные типы по умолчанию поддерживают ссылочную эквивалентность. В следующем примере ссылки `f1` и `f2` не являются эквивалентными — несмотря на то, что их объекты имеют идентичное содержимое:

```
class Foo { public int X; }
...
Foo f1 = new Foo { X = 5 };
Foo f2 = new Foo { X = 5 };
Console.WriteLine (f1 == f2);  // False
```

Напротив, `f3` и `f1` эквивалентны, поскольку они ссылаются на тот же самый объект:

```
Foo f3 = f1;
Console.WriteLine (f1 == f3);  // True
```

Позже в этом разделе мы объясним, каким образом можно *настраивать* ссылочные типы для обеспечения эквивалентности значений. Примером может служить класс `Uri` из пространства имен `System`:

```
Uri uri1 = new Uri ("http://www.linqpad.net");
Uri uri2 = new Uri ("http://www.linqpad.net");
Console.WriteLine (uri1 == uri2);  // True
```

## Стандартные протоколы эквивалентности

Существуют три стандартных протокола, которые типы могут внедрять для сравнения эквивалентности:

- операции == и !=;
- виртуальный метод `Equals` в классе `object`;
- интерфейс `IEquatable<T>`.

Вдобавок есть *подключаемые* протоколы и интерфейс `IStructuralEquatable`, который мы рассмотрим в главе 7.

### Операции == и !=

Вы уже видели в многочисленных примерах, каким образом стандартные операции == и != производят сравнения равенства/неравенства. Тонкости с == и != возникают из-за того, что они являются *операциями*, поэтому распознаются статически (на самом деле они реализованы в виде статических функций). Следовательно, когда вы используете операцию == или !=, то решение о том, какой тип будет выполнять сравнение, принимается *на этапе компиляции*, и виртуальное поведение в игру не вступает. Обычно подобное и желательно. В показанном далее примере компилятор жестко привязывает операцию == к типу `int`, т.к. переменные `x` и `y` имеют тип `int`:

```
int x = 5;
int y = 5;
Console.WriteLine (x == y);        // True
```

Но в представленном ниже примере компилятор привязывает операцию == к типу `object`:

```
object x = 5;
object y = 5;
Console.WriteLine (x == y);        // False
```

Поскольку `object` — класс (т.е. ссылочный тип), операция == типа `object` применяет для сравнения `x` и `y` *ссылочную эквивалентность*. Результатом будет `false`, потому что `x` и `y` ссылаются на разные упакованные объекты в куче.

### Виртуальный метод `Object.Equals`

Для корректного сравнения `x` и `y` в предыдущем примере мы можем использовать виртуальный метод `Equals`. Метод `Equals` определен в классе `System.Object`, поэтому он доступен всем типам:

```
object x = 5;
object y = 5;
Console.WriteLine (x.Equals (y)); // True
```

Метод `Equals` распознается во время выполнения — согласно действительному типу объекта. В данном случае вызывается метод `Equals` типа `Int32`, который приме-

няет к операндам *эквивалентность значений*, возвращая `true`. Со ссылочными типами метод `Equals` по умолчанию выполняет сравнение ссылочной эквивалентности; для структур метод `Equals` производит сравнение структурной эквивалентности, вызывая `Equals` на каждом их поле.

### Чем объясняется подобная сложность?

У вас может возникнуть вопрос, почему разработчики C# не попытались избежать проблемы, сделав операцию == виртуальной и таким образом функционально идентичной `Equals`? На то было несколько причин.

- Если первый операнд равен `null`, то метод `Equals` терпит неудачу с генерацией исключения `NullReferenceException`, а статическая операция – нет.
- Поскольку операция == распознается статически, она выполняется очень быстро. Это означает, что вы можете записывать код с интенсивными вычислениями без ущерба производительности – и без необходимости в изучении другого языка, такого как C++.
- Иногда полезно обеспечить для операции == и метода `Equals` разные определения эквивалентности. Мы опишем такой сценарий позже в разделе.

По существу сложность реализованного проектного решения отражает сложность самой ситуации: концепция эквивалентности охватывает большое число сценариев.

Следовательно, метод `Equals` подходит для сравнения двух объектов в независимой от типа манере. Показанный ниже метод сравнивает два объекта любого типа:

```
public static bool AreEqual (object obj1, object obj2)
  => obj1.Equals (obj2);
```

Тем не менее, есть один сценарий, при котором такой подход не срабатывает. Если первый аргумент равен `null`, то сгенерируется исключение `NullReferenceException`. Ниже приведена исправленная версия метода:

```
public static bool AreEqual (object obj1, object obj2)
{
  if (obj1 == null) return obj2 == null;
  return obj1.Equals (obj2);
}
```

Или более лаконично:

```
public static bool AreEqual (object obj1, object obj2)
  => obj1 == null ? obj2 == null : obj1.Equals (obj2);
```

### Статический метод `object.Equals`

В классе `object` определен статический вспомогательный метод, который выполняет работу метода `AreEqual` из предыдущего примера. Он имеет имя `Equals` (точно как у виртуального метода), но никаких конфликтов не возникает, т.к. он принимает *два* аргумента:

```
public static bool Equals (object objA, object objB)
```

Данный метод предоставляет алгоритм сравнения эквивалентности, безопасный к `null`, который предназначен для ситуаций, когда типы не известны на этапе компиляции. Например:

```
object x = 3, y = 3;
Console.WriteLine (object.Equals (x, y));  // True
x = null;
Console.WriteLine (object.Equals (x, y));  // False
y = null;
Console.WriteLine (object.Equals (x, y));  // True
```

Метод полезен при написании обобщенных типов. Приведенный ниже код не скомпилируется, если вызов `object.Equals` заменить операцией == или !=:

```
class Test <T>
{
  T _value;
  public void SetValue (T newValue)
  {
    if (!object.Equals (newValue, _value))
    {
      _value = newValue;
      OnValueChanged();
    }
  }
  protected virtual void OnValueChanged() { ... }
}
```

Операции здесь запрещены, потому что компилятор не может выполнить связывание со статическим методом неизвестного типа.

Более аккуратный способ реализации такого сравнения предусматривает использование класса `EqualityComparer<T>`. Преимущество заключается в том, что тогда удается избежать упаковки:

```
if (!EqualityComparer<T>.Default.Equals (newValue, _value))
```

Мы обсудим класс `EqualityComparer<T>` более подробно в разделе "Подключение протоколов эквивалентности и порядка" главы 7.

### Статический метод `object.ReferenceEquals`

Иногда требуется принудительно применять сравнение ссылочной эквивалентности. Статический метод `object.ReferenceEquals` делает именно это:

```
class Widget { ... }

class Test
{
  static void Main()
  {
    Widget w1 = new Widget();
    Widget w2 = new Widget();
    Console.WriteLine (object.ReferenceEquals (w1, w2));  // False
  }
}
```

Поступать так может быть необходимо из-за того, что в классе `Widget` возможно переопределение виртуального метода `Equals`, в результате чего `w1.Equals(w2)` будет возвращать `true`. Более того, в `Widget` возможна перегрузка операции == и сравнение `w1==w2` также будет давать `true`. В подобных случаях вызов `object.ReferenceEquals` гарантирует использование нормальной семантики ссылочной эквивалентности.

Еще один способ обеспечения сравнения ссылочной эквивалентности предусматривает приведение значений к `object` с последующим применением операции ==.

### Интерфейс `IEquatable<T>`

Последствием вызова метода `object.Equals` является упаковка типов значений. В сценариях с высокой критичностью к производительности это не желательно, т.к. по сравнению с действительным сравнением упаковка будет относительно затратной в плане ресурсов. Решение данной проблемы появилось в C# 2.0 — интерфейс `IEquatable<T>`:

```
public interface IEquatable<T>
{
  bool Equals (T other);
}
```

Идея в том, что реализация интерфейса `IEquatable<T>` обеспечивает такой же результат, как и вызов виртуального метода `Equals` из `object`, но только быстрее. Интерфейс `IEquatable<T>` реализован большинством базовых типов .NET. Интерфейс `IEquatable<T>` можно использовать как ограничение в обобщенном типе:

```
class Test<T> where T : IEquatable<T>
{
  public bool IsEqual (T a, T b)
  {
    return a.Equals (b);  // Упаковка с обобщенным типом T не происходит
  }
}
```

Если убрать ограничение обобщенного типа, то класс по-прежнему скомпилируется, но `a.Equals(b)` будет связываться с более медленным методом `object.Equals` (более медленным, исходя из предположения, что `T` — тип значения).

### Когда метод `Equals` и операция == не эквивалентны

Ранее мы упоминали, что временами для операции == и метода `Equals` полезно применять разные определения эквивалентности. Например:

```
double x = double.NaN;
Console.WriteLine (x == x);        // False
Console.WriteLine (x.Equals (x)); // True
```

Операция == в типе `double` гарантирует, что значение `NaN` никогда не может быть равным чему-либо еще — даже другому значению `NaN`. Это наиболее естественно с математической точки зрения и отражает внутреннее поведение центрального процессора. Однако метод `Equals` обязан использовать *рефлексивную* эквивалентность; другими словами, вызов `x.Equals(x)` *должен всегда возвращать* `true`.

На такое поведение `Equals` полагаются коллекции и словари; в противном случае они не смогут найти ранее сохраненный в них элемент.

Обеспечение отличающегося поведения эквивалентности в `Equals` и == для типов значений предпринимается довольно редко. Более распространенный сценарий относится к ссылочным типам и происходит, когда разработчик настраивает метод `Equals` так, что он реализует эквивалентность значений, оставляя операцию == вы-

полняющей (стандартную) ссылочную эквивалентность. Именно это делает класс StringBuilder:

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);        // False (ссылочная эквивалентность)
Console.WriteLine (sb1.Equals (sb2)); // True  (эквивалентность значений)
```

Теперь давайте посмотрим, как настраивать эквивалентность.

## Эквивалентность и специальные типы

Вспомним стандартное поведение сравнения эквивалентности:

- типы значений применяют *эквивалентность значений*;
- ссылочные типы используют *ссылочную эквивалентность*.

Кроме того:

- по умолчанию метод Equals структуры применяет *структурную эквивалентность значений* (т.е. сравниваются все поля в структурах).

При написании типа такое поведение иногда имеет смысл переопределять. Существуют две ситуации, когда требуется переопределение:

- для изменения смысла эквивалентности;
- для ускорения сравнений эквивалентности в структурах.

### Изменение смысла эквивалентности

Изменять смысл эквивалентности необходимо тогда, когда стандартное поведение операции == и метода Equals неестественно для типа и *не является тем поведением, которое будет ожидать потребитель*. Примером может служить DateTimeOffset – структура с двумя закрытыми полями: UTC-значение DateTime и целочисленное смещение. При написании подобного типа может понадобиться сделать так, чтобы сравнение эквивалентности принимало во внимание только поле с UTC-значением DateTime и не учитывало поле смещения. Другим примером являются числовые типы, поддерживающие значения NaN, вроде float и double. Если вы реализуете типы такого рода самостоятельно, то наверняка захотите обеспечить поддержку логики сравнения с NaN в сравнениях эквивалентности.

В случае классов иногда более естественно по умолчанию предлагать поведение *эквивалентности значений*, а не *ссылочной эквивалентности*. Это часто встречается в небольших классах, которые хранят простую порцию данных – например, System.Uri (или System.String).

### Ускорение сравнений эквивалентности для структур

Стандартный алгоритм сравнения *структурной эквивалентности* для структур является относительно медленным. Взяв на себя контроль над таким процессом за счет переопределения метода Equals, можно улучшить производительность в пять раз. Перегрузка операции == и реализация интерфейса IEquatable<T> делают возможными неупаковывающие сравнения эквивалентности, что также может ускорить производительность в пять раз.

Переопределение семантики эквивалентности для ссылочных типов не дает преимуществ в плане производительности. Стандартный алгоритм сравнения ссылочной эквивалентности уже отличается высокой скоростью, т.к. он просто сравнивает две 32- или 64-битные ссылки.

На самом деле существует другой, довольно специфический случай для настройки эквивалентности, который касается совершенствования алгоритма хеширования структуры в целях достижения лучшей производительности в хеш-таблице. Это проистекает из того факта, что сравнение эквивалентности и хеширование соединены в общий механизм. Хеширование рассматривается чуть позже в главе.

### Как переопределить семантику эквивалентности

Ниже представлена сводка по шагам.

1. Переопределите методы `GetHashCode` и `Equals`.
2. (Дополнительно) перегрузите операции `!=` и `==`.
3. (Дополнительно) реализуйте интерфейс `IEquatable<T>`.

### Переопределение метода `GetHashCode`

Может показаться странным, что в классе `System.Object` — с его небольшим набором членов — определен метод специализированного и узкого назначения. Такому описанию соответствует виртуальный метод `GetHashCode` в классе `Object`, который существует главным образом для извлечения выгоды следующими двумя типами:

```
System.Collections.Hashtable
System.Collections.Generic.Dictionary<TKey,TValue>
```

Они представляют собой *хеш-таблицы* — коллекции, в которых каждый элемент имеет ключ, используемый для сохранения и извлечения. В хеш-таблице применяется очень специфичная стратегия для эффективного выделения памяти под элементы на основе их ключей. Она требует, чтобы каждый ключ имел число типа `Int32`, или *хеш-код*. Хеш-код не обязан быть уникальным для каждого ключа, но должен быть насколько возможно разнообразным для достижения хорошей производительности хеш-таблицы. Хеш-таблицы считаются настолько важными, что метод `GetHashCode` определен в классе `System.Object`, а потому любой тип может выдавать хеш-код.

Хеш-таблицы детально описаны в разделе "Словари" главы 7.

Ссылочные типы и типы значений имеют стандартные реализации метода `GetHashCode`, т.е. переопределять данный метод не придется — *если только не переопределяется* метод `Equals`. (И если вы переопределяете метод `GetHashCode`, то почти наверняка захотите также переопределить метод `Equals`.) Существуют и другие правила, касающиеся переопределения метода `object.GetHashCode`.

- Он должен возвращать одно и то же значение на двух объектах, для которых метод `Equals` возвращает `true` (поэтому `GetHashCode` и `Equals` переопределяются вместе).
- Он не должен генерировать исключения.
- Он должен возвращать одно и то же значение при многократных вызовах на том же самом объекте (если только объект не *изменился*).

Для достижения максимальной производительности в хеш-таблицах метод GetHashCode должен быть написан так, чтобы минимизировать вероятность того, что два разных значения получат один и тот же хеш-код. Это порождает третью причину переопределения методов Equals и GetHashCode в структурах – предоставление алгоритма хеширования, который эффективнее стандартного. Стандартная реализация для структур возлагается на исполняющую среду и может основываться на каждом поле в структуре.

Напротив, стандартная реализация метода GetHashCode для классов основана на внутреннем маркере объекта, который уникален для каждого экземпляра в текущей реализации CLR.

Если хеш-код объекта изменяется после того, как он был добавлен как ключ в словарь, то объект больше не будет доступен в словаре. Такую ситуацию можно предотвратить путем базирования вычислений хеш-кодов на неизменяемых полях.

Вскоре мы рассмотрим завершенный пример, иллюстрирующий переопределение метода GetHashCode.

### Переопределение метода Equals

Ниже перечислены постулаты, касающиеся метода object.Equals.

- Объект не может быть эквивалентен null (если только он не относится к типу, допускающему значение null).
- Эквивалентность *рефлексивна* (объект эквивалентен самому себе).
- Эквивалентность *симметрична* (если a.Equals(b), то b.Equals(a)).
- Эквивалентность *транзитивна* (если a.Equals(b) и b.Equals(c), то a.Equals(c)).
- Операции эквивалентности повторяемы и надежны (они не генерируют исключений).

### Перегрузка операций == и !=

В дополнение к переопределению метода Equals можно необязательно перегрузить операции == и !=. Так почти всегда поступают для структур, потому что в противном случае операции == и != просто не будут работать для типа.

В случае классов возможны два пути:

- оставить операции == и != незатронутыми – тогда они будут использовать ссылочную эквивалентность;
- перегрузить == и != вместе с Equals.

Первый подход чаще всего применяется в специальных типах – особенно в *изменяемых* типах. Он гарантирует ожидаемое поведение, заключающееся в том, что операции == и != должны обеспечивать ссылочную эквивалентность со ссылочными типами, и позволяет избежать путаницы у потребителей специальных типов. Пример уже приводился ранее:

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);       // False (ссылочная эквивалентность)
Console.WriteLine (sb1.Equals (sb2)); // True  (эквивалентность значений)
```

Второй подход имеет смысл использовать с типами, для которых потребителю никогда не потребуется ссылочная эквивалентность. Такие типы обычно неизменяемы — как классы `string` и `System.Uri` — и временами являются хорошими кандидатами на реализацию в виде структур.

Хотя возможна перегрузка операции `!=` с приданием ей смысла, отличающегося от `!(==)`, на практике так почти никогда не поступают за исключением случаев, подобных сравнению с `float.NaN`.

### Реализация интерфейса `IEquatable<T>`

Для полноты при переопределении метода `Equals` также неплохо реализовать интерфейс `IEquatable<T>`. Его результаты должны всегда соответствовать результатам переопределенного метода `Equals` класса `object`. Реализация `IEquatable<T>` не требует никаких усилий по программированию, если реализация метода `Equals` структурирована, как демонстрируется в следующем примере.

### Пример: структура `Area`

Предположим, что необходима структура для представления области, ширина и высота которой взаимозаменяемы. Другими словами, 5 × 10 эквивалентно 10 × 5. (Такой тип был бы подходящим в алгоритме упорядочения прямоугольных форм.)

Ниже приведен полный код:

```
public struct Area : IEquatable <Area>
{
  public readonly int Measure1;
  public readonly int Measure2;

  public Area (int m1, int m2)
  {
    Measure1 = Math.Min (m1, m2);
    Measure2 = Math.Max (m1, m2);
  }

  public override bool Equals (object other)
  {
    if (!(other is Area)) return false;
    return Equals ((Area) other);  // Вызывает метод, определенный ниже
  }

  public bool Equals (Area other)  // Реализует IEquatable<Area>
    => Measure1 == other.Measure1 && Measure2 == other.Measure2;

  public override int GetHashCode()
    => Measure2 * 31 + Measure1;   // 31 - некоторое простое число

  public static bool operator == (Area a1, Area a2) => a1.Equals (a2);

  public static bool operator != (Area a1, Area a2) => !a1.Equals (a2);
}
```

Вот другой способ реализации метода `Equals`, в котором задействованы типы, допускающие `null`:

```
Area? otherArea = other as Area?;
return otherArea.HasValue && Equals (otherArea.Value);
```

В реализации метода GetHashCode мы поспособствовали увеличению вероятности гарантии уникальности за счет того, что перед сложением двух размеров умножили больший размер на некоторое простое число (игнорируя переполнение). При наличии более двух полей следующий паттерн, предложенный Джошуа Блохом, обеспечивает хорошие результаты с приемлемой производительностью:

```
int hash = 17;                                  // 17 - некоторое простое число
hash = hash * 31 + field1.GetHashCode();        // 31 - еще одно простое число
hash = hash * 31 + field2.GetHashCode();
hash = hash * 31 + field3.GetHashCode();
...
return hash;
```

(Ссылку на обсуждение простых чисел и хеш-кодов ищите по адресу http://albahari.com/hashprimes.)

Ниже демонстрируется применение структуры Area:

```
Area a1 = new Area (5, 10);
Area a2 = new Area (10, 5);
Console.WriteLine (a1.Equals (a2));     // True
Console.WriteLine (a1 == a2);           // True
```

### Подключаемые компараторы эквивалентности

Если необходимо, чтобы тип задействовал другую семантику эквивалентности только в конкретном сценарии, то можно воспользоваться подключаемым компаратором эквивалентности IEqualityComparer. Он особенно удобен в сочетании со стандартными классами коллекций, и мы рассмотрим такие вопросы в разделе "Подключение протоколов эквивалентности и порядка" главы 7.

## Сравнение порядка

Помимо определения стандартных протоколов для эквивалентности в C# и .NET также предусмотрены стандартные протоколы для определения порядка, в котором один объект соотносится с другим. Базовые протоколы таковы:

- интерфейсы IComparable (IComparable и IComparable<T>);
- операции > и <.

Интерфейсы IComparable применяются универсальными алгоритмами сортировки. В следующем примере статический метод Array.Sort работает по той причине, что класс System.String реализует интерфейсы IComparable:

```
string[] colors = { "Green", "Red", "Blue" };
Array.Sort (colors);
foreach (string c in colors) Console.Write (c + " ");  // Blue Green Red
```

Операции < и > более специализированы и предназначены в основном для числовых типов. Поскольку эти операции распознаются статически, они могут транслироваться в высокоэффективный байт-код, подходящий для алгоритмов с интенсивными вычислениями.

Инфраструктура .NET Framework также предлагает подключаемые протоколы упорядочения через интерфейсы IComparer. Мы опишем их в финальном разделе главы 7.

## Интерфейсы IComparable

Интерфейсы IComparable определены следующим образом:

```
public interface IComparable       { int CompareTo (object other); }
public interface IComparable<in T> { int CompareTo (T other);      }
```

Указанные два интерфейса представляют ту же самую функциональность. Для типов значений обобщенный интерфейс, безопасный к типам, оказывается быстрее, чем необобщенный интерфейс. В обоих случаях метод CompareTo работает так, как описано ниже:

- если a находится после b, то a.CompareTo(b) возвращает положительное число;
- если a такое же, как и b, то a.CompareTo(b) возвращает 0;
- если a находится перед b, то a.CompareTo(b) возвращает отрицательное число.

Например:

```
Console.WriteLine ("Beck".CompareTo ("Anne"));   // 1
Console.WriteLine ("Beck".CompareTo ("Beck"));   // 0
Console.WriteLine ("Beck".CompareTo ("Chris"));  // -1
```

Большинство базовых типов реализуют оба интерфейса IComparable. Эти интерфейсы также иногда реализуются при написании специальных типов. Вскоре мы рассмотрим пример.

### IComparable или Equals

Предположим, что в некотором типе переопределен метод Equals и тип также реализует интерфейсы IComparable. Вы ожидаете, что когда метод Equals возвращает true, то метод CompareTo должен возвращать 0. И будете правы. Но вот в чем загвоздка:

- когда Equals возвращает false, метод CompareTo может возвращать то, что ему заблагорассудится (до тех пор, пока это внутренне непротиворечиво)!

Другими словами, эквивалентность может быть "придирчивее", чем сравнение, но не наоборот (стоит правило нарушить – и алгоритмы сортировки перестанут работать). Таким образом, метод CompareTo может сообщать: "Все объекты равны", тогда как метод Equals сообщает: "Но некоторые из них более равны, чем другие".

Великолепным примером может служить класс System.String. Метод Equals и операция == в классе String используют *ординальное* сравнение, при котором сравниваются значения кодовых знаков Unicode каждого символа. Однако метод CompareTo применяет менее придирчивое сравнение, зависимое от культуры. Скажем, на большинстве компьютеров строки "ü" и "ü" будут разными согласно Equals, но одинаковыми согласно CompareTo.

В главе 7 мы обсудим подключаемый протокол упорядочения IComparer, который позволяет указывать альтернативный алгоритм упорядочения при сортировке или при создании экземпляра сортированной коллекции. Специальная реализация IComparer может и дальше расширять разрыв между CompareTo и Equals – например, нечувствительный к регистру компаратор строк будет возвращать 0 в качестве результата сравнения "A" и "a". Тем не менее, обратное правило по-прежнему применимо: метод CompareTo никогда не может быть придирчивее, чем Equals.

При реализации интерфейсов `IComparable` в специальном типе нарушения данного правила можно избежать, поместив в первую строку метода `CompareTo` следующий оператор:

```
if (Equals (other)) return 0;
```

После этого можно возвращать то, что нравится, при условии соблюдения согласованности!

## Операции < и >

В некоторых типах определены операции < и >. Например:

```
bool after2010 = DateTime.Now > new DateTime (2010, 1, 1);
```

Можно ожидать, что операции < и >, когда они реализованы, должны быть функционально согласованными с интерфейсами `IComparable`. Такая стандартная практика повсеместно соблюдается в .NET Framework.

Также стандартной практикой является реализация интерфейсов `IComparable` всякий раз, когда операции < и > перегружаются, хотя обратное утверждение неверно. В действительности большинство типов .NET, реализующих `IComparable`, *не* перегружают операции < и >. Это отличается от ситуации с эквивалентностью, при которой в случае переопределения метода `Equals` обычно перегружается операция ==.

Как правило, операции > и < перегружаются только в перечисленных ниже случаях.

- Тип обладает строгой внутренне присущей концепцией "больше чем" и "меньше чем" (в противоположность более широким концепциям "находится перед" и "находится после" интерфейса `IComparable`).
- Существует только один способ *или контекст*, в котором производится сравнение.
- Результат инвариантен для различных культур.

Класс `System.String` не удовлетворяет последнему пункту: результаты сравнения строк в разных языках могут варьироваться. Следовательно, тип `string` не поддерживает операции > и <:

```
bool error = "Beck" > "Anne";  // Ошибка на этапе компиляции
```

## Реализация интерфейсов `IComparable`

В следующей структуре, представляющей музыкальную ноту, мы реализуем интерфейсы `IComparable`, а также перегружаем операции < и >. Для полноты мы также переопределяем методы `Equals`/`GetHashCode` и перегружаем операции == и !=:

```
public struct Note : IComparable<Note>, IEquatable<Note>, IComparable
{
  int _semitonesFromA;
  public int SemitonesFromA { get { return _semitonesFromA; } }

  public Note (int semitonesFromA)
  {
    _semitonesFromA = semitonesFromA;
  }

  public int CompareTo (Note other) // Обобщенный интерфейс IComparable<T>
  {
    if (Equals (other)) return 0;    // Отказоустойчивая проверка
    return _semitonesFromA.CompareTo (other._semitonesFromA);
  }
```

```
  int IComparable.CompareTo (object other) //Необобщенный интерфейс IComparable
  {
    if (!(other is Note))
      throw new InvalidOperationException ("CompareTo: Not a note"); //не нота
    return CompareTo ((Note) other);
  }
  public static bool operator < (Note n1, Note n2)
     => n1.CompareTo (n2) < 0;
  public static bool operator > (Note n1, Note n2)
    => n1.CompareTo (n2) > 0;
  public bool Equals (Note other)  // для IEquatable<Note>
    => _semitonesFromA == other._semitonesFromA;
  public override bool Equals (object other)
  {
    if (!(other is Note)) return false;
    return Equals ((Note) other);
  }
  public override int GetHashCode() => _semitonesFromA.GetHashCode();
  public static bool operator == (Note n1, Note n2) => n1.Equals (n2);
  public static bool operator != (Note n1, Note n2) => !(n1 == n2);
}
```

# Служебные классы

## Класс Console

Статический класс Console обрабатывает ввод-вывод для консольных приложений. В приложении командной строки (консольном приложении) ввод поступает с клавиатуры через методы Read, ReadKey и ReadLine, а вывод осуществляется в текстовое окно посредством методов Write и WriteLine. Управлять позицией и размерами такого окна можно с помощью свойств WindowLeft, WindowTop, WindowHeight и WindowWidth. Можно также изменять свойства BackgroundColor и ForegroundColor и манипулировать курсором через свойства CursorLeft, CursorTop и CursorSize:

```
Console.WindowWidth = Console.LargestWindowWidth;
Console.ForegroundColor = ConsoleColor.Green;
Console.Write ("test... 50%");
Console.CursorLeft -= 3;
Console.Write ("90%");  // test... 90%
```

Методы Write и WriteLine перегружены для приема смешанной форматной строки (см. раздел "Метод String.Format и смешанные форматные строки" ранее в главе). Тем не менее, ни один из методов не принимает поставщик формата, так что вы привязаны к CultureInfo.CurrentCulture. (Разумеется, существует обходной путь, предусматривающий явный вызов string.Format.)

Свойство Console.Out возвращает объект TextWriter. Передача Console.Out методу, который ожидает TextWriter — удобный способ заставить этот метод выводить что-либо на консоль в целях диагностики.

Можно также перенаправлять потоки ввода и вывода на консоль с помощью методов SetIn и SetOut:

```
// Сначала сохранить существующий объект записи вывода:
System.IO.TextWriter oldOut = Console.Out;

// Перенаправить консольный вывод в файл:
using (System.IO.TextWriter w = System.IO.File.CreateText("e:\\output.txt"))
{
  Console.SetOut (w);
  Console.WriteLine ("Hello world");
}

// Восстановить стандартный консольный вывод:
Console.SetOut (oldOut);

// Открыть файл output.txt в Блокноте:
System.Diagnostics.Process.Start ("e:\\output.txt");
```

Работа потоков данных и объектов записи текста обсуждается в главе 15.

При запуске приложений WPF и Windows Forms в среде Visual Studio консольный вывод автоматически перенаправляется в окно вывода Visual Studio (в режиме отладки). Такая особенность делает метод `Console.Write` полезным для диагностических целей, хотя в большинстве случаев более подходящими будут классы `Debug` и `Trace` из пространства имен `System.Diagnostics` (глава 13).

## Класс `Environment`

Статический класс `System.Environment` предлагает набор полезных свойств, предназначенных для взаимодействия с разнообразными сущностями.

**Файлы и папки**

`CurrentDirectory`, `SystemDirectory`, `CommandLine`

**Компьютер и операционная система**

`MachineName`, `ProcessorCount`, `OSVersion`, `NewLine`

**Пользователь, вошедший в систему**

`UserName`, `UserInteractive`, `UserDomainName`

**Диагностика**

`TickCount`, `StackTrace`, `WorkingSet`, `Version`

Дополнительные папки можно получить путем вызова метода `GetFolderPath`; мы рассмотрим это в разделе "Операции с файлами и каталогами" главы 15.

Обращаться к переменным среды операционной системы (которые просматриваются за счет ввода `set` в командной строке) можно с помощью следующих трех методов: `GetEnvironmentVariable`, `GetEnvironmentVariables` и `SetEnvironmentVariable`.

Свойство `ExitCode` позволяет установить код возврата, предназначенный для ситуации, когда программа запускается из команды либо из пакетного файла, а метод `FailFast` завершает программу немедленно, не выполняя очистку.

Класс `Environment`, доступный приложениям Windows Store, предлагает только ограниченное количество членов (`ProcessorCount`, `NewLine` и `FailFast`).

## Класс Process

Класс Process из пространства имен System.Diagnostics позволяет запускать новый процесс.

Статический метод Process.Start имеет несколько перегруженных версий; простейшая из них принимает имя файла с необязательными аргументами:

```
Process.Start ("notepad.exe");
Process.Start ("notepad.exe", "e:\\file.txt");
```

Можно также указать только имя файла, и тогда запустится программа, зарегистрированная для файлового расширения:

```
Process.Start ("e:\\file.txt");
```

Наиболее гибкая перегруженная версия принимает экземпляр ProcessStartInfo. С его помощью можно захватывать и перенаправлять ввод, вывод и вывод ошибок запущенного процесса (если свойство UseShellExecute установлено в false). В следующем коде захватывается вывод утилиты ipconfig:

```
ProcessStartInfo psi = new ProcessStartInfo
{
  FileName = "cmd.exe",
  Arguments = "/c ipconfig /all",
  RedirectStandardOutput = true,
  UseShellExecute = false
};
Process p = Process.Start (psi);
string result = p.StandardOutput.ReadToEnd();
Console.WriteLine (result);
```

То же самое можно сделать для вызова компилятора csc, установив Filename, как показано ниже:

```
psi.FileName = System.IO.Path.Combine (
  System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory(),
  "csc.exe");
```

Если вывод не перенаправлен, то метод Process.Start выполняет программу параллельно вызывающей программе. Если нужно ожидать завершения нового процесса, тогда можно вызвать метод WaitForExit на объекте Process с указанием необязательного тайм-аута.

Класс Process также позволяет запрашивать и взаимодействовать с другими процессами, функционирующими на компьютере (глава 13).

По причинам, связанным с безопасностью, класс Process не доступен приложениям Windows Store, поэтому запускать произвольные процессы невозможно. Взамен должен использоваться класс Windows.System.Launcher для "запуска" URI или файла, к которому имеется доступ, например:

```
Launcher.LaunchUriAsync (new Uri ("http://albahari.com"));
var file = await KnownFolders.DocumentsLibrary.GetFileAsync ("foo.txt");
Launcher.LaunchFileAsync (file);
```

Такой код приводит к открытию URI или файла с применением любой программы, ассоциированной со схемой URI или файловым расширением. Чтобы все работало, программа должна функционировать на переднем плане.

## Класс AppContext

Класс System.AppContext появился в версии .NET Framework 4.6. Он представляет глобальный словарь булевских значений со строковыми ключами и предназначен для разработчиков библиотек в качестве стандартного механизма, который позволяет потребителям включать и отключать новые функциональные средства. Этот нетипизированный подход имеет смысл использовать с экспериментальными функциональными средствами, которые желательно сохранять недокументированными для большинства пользователей.

Потребитель библиотеки требует, чтобы определенное функциональное средство было включено, следующим образом:

```
AppContext.SetSwitch ("MyLibrary.SomeBreakingChange", true);
```

Затем код внутри библиотеки может проверять признак включения функционального средства:

```
bool isDefined, switchValue;
isDefined = AppContext.TryGetSwitch ("MyLibrary.SomeBreakingChange",
                                     out switchValue);
```

Метод TryGetSwitch возвращает false, если признак включения не определен; это позволяет различать неопределенный признак включения от признака, значение которого установлено в false, когда в таком действии возникнет необходимость.

По иронии судьбы проектное решение, положенное в основу метода TryGetSwitch, иллюстрирует то, как не следует разрабатывать API-интерфейсы. Параметр out является излишним, а взамен метод должен возвращать допускающий null тип bool, который принимает значение true, false или null для неопределенного признака включения. В таком случае его можно было бы применять следующим образом:

```
bool switchValue = AppContext.GetSwitch ("...") ?? false;
```

# 7

# Коллекции

Инфраструктура .NET Framework предоставляет стандартный набор типов для хранения и управления коллекциями объектов. Сюда входят списки с изменяемыми размерами, связные списки, отсортированные и несортированные словари и массивы. Из всего перечисленного только массивы являются частью языка C#; остальные коллекции – это просто классы, экземпляры которых можно создавать подобно любым другим классам.

Типы в .NET Framework для коллекций могут быть разделены на следующие категории:

- интерфейсы, которые определяют стандартные протоколы коллекций;
- готовые к использованию классы коллекций (списки, словари и т.д.);
- базовые классы, позволяющие создавать коллекции, специфичные для приложений.

В настоящей главе рассматриваются все указанные категории, а также типы, применяемые при определении эквивалентности и порядка следования элементов.

Ниже перечислены пространства имен коллекций.

| Пространство имен | Что содержит |
|---|---|
| `System.Collections` | Необобщенные классы и интерфейсы коллекций |
| `System.Collections.Specialized` | Строго типизированные необобщенные классы коллекций |
| `System.Collections.Generic` | Обобщенные классы и интерфейсы коллекций |
| `System.Collections.ObjectModel` | Прокси и базовые классы для специальных коллекций |
| `System.Collections.Concurrent` | Коллекции, безопасные к потокам (глава 23) |

## Перечисление

В программировании существует много разнообразных коллекций, начиная с простых структур данных вроде массивов и связных списков и заканчивая сложными структурами, такими как красно-черные деревья и хеш-таблицы. Хотя внутренняя

реализация и внешние характеристики таких структур данных варьируются в широких пределах, возможность прохода по содержимому коллекции является наиболее универсальной потребностью. Инфраструктура .NET Framework поддерживает эту потребность через пару интерфейсов (`IEnumerable`, `IEnumerator` и их обобщенные аналоги), которые позволяют различным структурам данных открывать доступ к общим API-интерфейсам обхода. Они представляют собой часть более крупного множества интерфейсов коллекций, которые показаны на рис. 7.1.

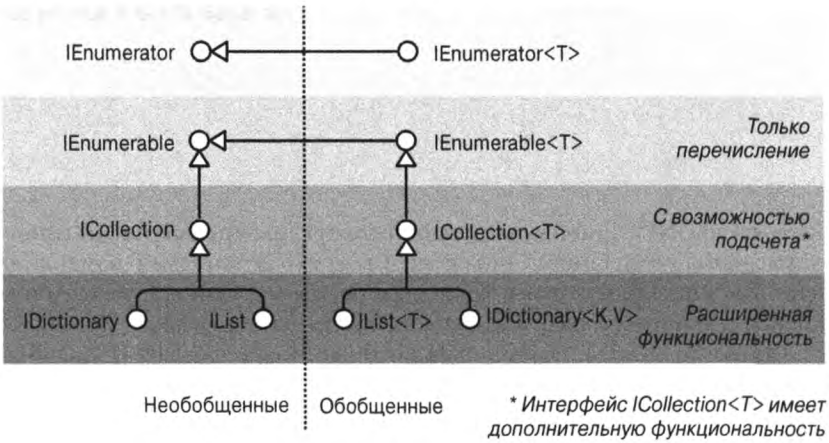



***Рис. 7.1.** Интерфейсы коллекций*

## `IEnumerable` и `IEnumerator`

Интерфейс `IEnumerator` определяет базовый низкоуровневый протокол, посредством которого производится проход по элементам – или перечисление – коллекции в однонаправленной манере. Объявление этого интерфейса показано ниже:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Метод `MoveNext` передвигает текущий элемент, или "курсор", на следующую позицию, возвращая `false`, если в коллекции больше не осталось элементов. Метод `Current` возвращает элемент в текущей позиции (обычно приводя его от `object` к более специфичному типу). Перед извлечением первого элемента должен быть вызван метод `MoveNext`, что нужно для учета пустой коллекции. Метод `Reset`, когда он реализован, выполняет перемещение к началу, делая возможным перечисление коллекции заново. Метод `Reset` существует главным образом для взаимодействия с COM: вызова его напрямую в общем случае избегают, т.к. он не является универсально поддерживаемым (и он необязателен, потому что обычно просто создает новый экземпляр перечислителя).

Как правило, коллекции не *реализуют* перечислители; взамен они *предоставляют* их через интерфейс `IEnumerable`:

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

За счет определения единственного метода, возвращающего перечислитель, интерфейс `IEnumerable` обеспечивает гибкость в том, что реализация логики итерации может быть возложена на другой класс. Кроме того, это означает, что несколько потребителей способны выполнять перечисление коллекции одновременно, не влияя друг на друга. Интерфейс `IEnumerable` можно воспринимать как "поставщика `IEnumerator`", и он является наиболее базовым интерфейсом, который реализуют классы коллекций.

В следующем примере демонстрируется низкоуровневое использование интерфейсов `IEnumerable` и `IEnumerator`:

```
string s = "Hello";

// Так как тип string реализует IEnumerable, мы можем вызывать GetEnumerator():
IEnumerator rator = s.GetEnumerator();

while (rator.MoveNext())
{
  char c = (char) rator.Current;
  Console.Write (c + ".");
}

// Вывод: H.e.l.l.o.
```

Однако вызов методов перечислителей напрямую в такой манере встречается редко, поскольку C# предоставляет синтаксическое сокращение: оператор `foreach`. Ниже приведен тот же самый пример, переписанный с применением `foreach`:

```
string s = "Hello";  // Класс String реализует интерфейс IEnumerable

foreach (char c in s)
  Console.Write (c + ".");
```

## IEnumerable<T> и IEnumerator<T>

Интерфейсы `IEnumerator` и `IEnumerable` почти всегда реализуются в сочетании со своими расширенными обобщенными версиями:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
  T Current { get; }
}
public interface IEnumerable<T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
```

Определяя типизированные версии свойства `Current` и метода `GetEnumerator`, данные интерфейсы усиливают статическую безопасность типов, избегают накладных расходов на упаковку элементов типов значений и повышают удобство эксплуатации потребителями. Массивы автоматически реализуют `IEnumerable<T>` (где `T` — тип элементов массива).

Благодаря улучшенной статической безопасности типов вызов следующего метода с массивом символов приведет к возникновению ошибки на этапе компиляции:

```
void Test (IEnumerable<int> numbers) { ... }
```

Для классов коллекций стандартной практикой является открытие общего доступа к IEnumerable<T> и в то же время "сокрытие" необобщенного IEnumerable посредством явной реализации интерфейса. Это значит, что в случае вызова напрямую метода GetEnumerator обратно получается реализация безопасного к типам обобщенного интерфейса IEnumerator<T>. Тем не менее, временами такое правило нарушается из-за необходимости обеспечения обратной совместимости (до версии C# 2.0 обобщения не существовали). Хорошим примером считаются массивы — во избежание нарушения работы ранее написанного кода они должны возвращать экземпляр необобщенного интерфейса IEnumerator. Для того чтобы получить обобщенный интерфейс IEnumerator<T>, придется выполнить явное приведение к соответствующему интерфейсу:

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable <int>)data).GetEnumerator();
```

К счастью, благодаря наличию оператора foreach писать код подобного рода приходится редко.

### IEnumerable<T> и IDisposable

Интерфейс IEnumerator<T> унаследован от IDisposable. Это позволяет перечислителям хранить ссылки на такие ресурсы, как подключения к базам данных, и гарантировать, что ресурсы будут освобождены, когда перечисление завершится (или прекратится на полпути). Оператор foreach распознает такие детали и транслирует код:

```
foreach (var element in somethingEnumerable) { ... }
```

в следующий логический эквивалент:

```
using (var rator = somethingEnumerable.GetEnumerator())
  while (rator.MoveNext())
  {
    var element = rator.Current;
    ...
  }
```

Блок using обеспечивает освобождение (более подробно об интерфейсе IDisposable речь пойдет в главе 12).

### Когда необходимо использовать необобщенные интерфейсы?

С учетом дополнительной безопасности типов, присущей обобщенным интерфейсам коллекций вроде IEnumerable<T>, возникает вопрос: нужно ли вообще применять необобщенный интерфейс IEnumerable (ICollection или IList)?

В случае интерфейса IEnumerable вы должны реализовать его в сочетании с IEnumerable<T>, т.к. последний является производным от первого. Однако вы очень редко будете действительно реализовывать данные интерфейсы с нуля: почти во всех ситуациях можно принять высокоуровневый подход, предусматривающий использование методов итератора, Collection<T> и LINQ.

А что насчет потребителя? Практически во всех случаях управление может осуществляться целиком с помощью обобщенных интерфейсов. Тем не менее, необобщенные интерфейсы по-прежнему иногда полезны своей способностью обеспечивать унификацию типов для коллекций по всем типам элементов. Например, показанный ниже метод подсчитывает элементы в любой коллекции *рекурсивным образом*:

```
public static int Count (IEnumerable e)
{
  int count = 0;
  foreach (object element in e)
  {
    var subCollection = element as IEnumerable;
    if (subCollection != null)
      count += Count (subCollection);
    else
      count++;
  }
  return count;
}
```

Поскольку язык C# предлагает ковариантность с обобщенными интерфейсами, может показаться допустимым заставить данный метод взамен принимать тип `IEnumerable<object>`. Однако тогда метод потерпит неудачу с элементами типов значений и унаследованными коллекциями, которые не реализуют интерфейс `IEnumerable<T>` — примером может служить класс `ControlCollection` из Windows Forms.

Кстати, в приведенном примере вы могли заметить потенциальную ошибку: *циклические* ссылки приведут к бесконечной рекурсии и аварийному отказу метода. Устранить проблему проще всего за счет применения типа `HashSet` (см. раздел "`HashSet<T>` и `SortedSet<T>`" далее в главе).

## Реализация интерфейсов перечисления

Реализация интерфейса `IEnumerable` или `IEnumerable<T>` может понадобиться по одной или нескольким описанным ниже причинам:

- для поддержки оператора `foreach`;
- для взаимодействия со всем, что ожидает стандартной коллекции;
- для удовлетворения требований более развитого интерфейса коллекции;
- для поддержки инициализаторов коллекций.

Чтобы реализовать `IEnumerable/IEnumerable<T>`, потребуется предоставить перечислитель. Это можно сделать одним из трех способов:

- если класс является оболочкой другой коллекции, то путем возвращения перечислителя внутренней коллекции;
- через итератор с использованием оператора `yield return`;
- за счет создания экземпляра собственной реализации `IEnumerator/IEnumerator<T>`.

Можно также создать подкласс существующей коллекции: класс `Collection<T>` предназначен как раз для такой цели (см. раздел "Настраиваемые коллекции и прокси" далее в главе). Еще один подход предусматривает применение операций запросов LINQ, которые рассматриваются в следующей главе.

Возвращение перечислителя другой коллекции сводится к вызову метода GetEnumerator на внутренней коллекции. Однако данный прием жизнеспособен только в простейших сценариях, где элементы во внутренней коллекции являются в точности тем, что требуется. Более гибкий подход заключается в написании итератора с использованием оператора yield return. *Итератор* – это средство языка C#, которое помогает в написании коллекций таким же способом, как оператор foreach оказывает содействие в их потреблении. Итератор автоматически поддерживает реализацию интерфейсов IEnumerable и IEnumerator либо их обобщенных версий. Ниже приведен простой пример:

```
public class MyCollection : IEnumerable
{
  int[] data = { 1, 2, 3 };

  public IEnumerator GetEnumerator()
  {
    foreach (int i in data)
      yield return i;
  }
}
```

Обратите внимание на "черную магию": кажется, что метод GetEnumerator вообще не возвращает перечислитель! Столкнувшись с оператором yield return, компилятор "за кулисами" генерирует скрытый вложенный класс перечислителя, после чего проводит рефакторинг метода GetEnumerator для создания и возвращения экземпляра данного класса. Итераторы отличаются мощью и простотой (и широко применяются в реализации стандартных операций запросов LINQ to Objects).

Придерживаясь такого подхода, мы можем также реализовать обобщенный интерфейс IEnumerable<T>:

```
public class MyGenCollection : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };

  public IEnumerator<int> GetEnumerator()
  {
    foreach (int i in data)
      yield return i;
  }

  IEnumerator IEnumerable.GetEnumerator()   // Явная реализация
  {                                         // сохраняет его скрытым
    return GetEnumerator();
  }
}
```

Из-за того, что интерфейс IEnumerable<T> унаследован от IEnumerable, мы должны реализовывать как обобщенную, так и необобщенную версии метода GetEnumerator. В соответствии со стандартной практикой мы реализовали необобщенную версию явно. Она просто вызывает обобщенную версию GetEnumerator, поскольку интерфейс IEnumerator<T> унаследован от IEnumerator.

Только что написанный класс подошел бы в качестве основы для написания более развитой коллекции. Тем не менее, если ничего сверх простой реализации интерфейса IEnumerable<T> не требуется, то оператор yield return предлагает упрощенный вариант. Вместо написания класса логику итерации можно переместить внутрь метода, возвращающего обобщенную реализацию IEnumerable<T>, и позволить компилятору позаботиться об остальном.

Например:

```
public class Test
{
  public static IEnumerable <int> GetSomeIntegers()
  {
    yield return 1;
    yield return 2;
    yield return 3;
  }
}
```

А вот как такой метод используется:

```
foreach (int i in Test.GetSomeIntegers())
  Console.WriteLine (i);

// Вывод
1
2
3
```

Последний подход к написанию метода `GetEnumerator` предполагает построение класса, который реализует интерфейс `IEnumerator` напрямую. Это в точности то, что компилятор делает "за кулисами", когда распознает итераторы. (К счастью, заходить настолько далеко вам придется редко.) В следующем примере определяется коллекция, содержащая целые числа 1, 2 и 3:

```
public class MyIntList : IEnumerable
{
  int[] data = { 1, 2, 3 };

  public IEnumerator GetEnumerator()
  {
    return new Enumerator (this);
  }

  class Enumerator : IEnumerator   // Определить внутренний
  {                                // класс для перечислителя
    MyIntList collection;
    int currentIndex = -1;

    public Enumerator (MyIntList collection)
    {
      this.collection = collection;
    }

    public object Current
    {
      get
      {
        if (currentIndex == -1)
          throw new InvalidOperationException ("Enumeration not started!");
                                              // Перечисление не началось!
        if (currentIndex == collection.data.Length)
          throw new InvalidOperationException ("Past end of list!");
                                             // Пройден конец списка!
        return collection.data [currentIndex];
      }
    }
```

```
    public bool MoveNext()
    {
      if (currentIndex >= collection.data.Length - 1) return false;
      return ++currentIndex < collection.data.Length;
    }
    public void Reset() { currentIndex = -1; }
  }
}
```

Реализовывать метод `Reset` вовсе не обязательно — взамен можно сгенерировать исключение `NotSupportedException`.

Обратите внимание, что первый вызов `MoveNext` должен переместить на первый (а не на второй) элемент в списке.

Чтобы сравняться с итератором в плане функциональности, мы должны также реализовать интерфейс `IEnumerator<T>`. Ниже приведен пример, в котором для простоты опущена проверка границ:

```
class MyIntList : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };
  // Обобщенный перечислитель совместим как с IEnumerable, так и с IEnumerable<T>.
  // Во избежание конфликта имен мы реализуем
  // необобщенный метод GetEnumerator явно.
  public IEnumerator<int> GetEnumerator() { return new Enumerator(this); }
  IEnumerator IEnumerable.GetEnumerator() { return new Enumerator(this); }
  class Enumerator : IEnumerator<int>
  {
    int currentIndex = -1;
    MyIntList collection;
    public Enumerator (MyIntList collection)
    {
      this.collection = collection;
    }
    public int Current => collection.data [currentIndex];
    object IEnumerator.Current => Current;
    public bool MoveNext() => ++currentIndex < collection.data.Length;
    public void Reset() => currentIndex = -1;
    // С учетом того, что метод Dispose не нужен, рекомендуется реализовать
    // его явно, чтобы он не был виден через открытый интерфейс.
    void IDisposable.Dispose() {}
  }
}
```

Пример с обобщениями функционирует быстрее, т.к. свойство `IEnumerator<int>.Current` не требует приведения `int` к `object`, что позволяет избежать накладных расходов, связанных с упаковкой.

# Интерфейсы `ICollection` и `IList`

Хотя интерфейсы перечисления предлагают протокол для однонаправленной итерации по коллекции, они не предоставляют механизма, который бы позволил опре-

делять размер коллекции, получать доступ к членам по индексу, производить поиск или модифицировать коллекцию. Для такой функциональности в .NET Framework определены интерфейсы `ICollection`, `IList` и `IDictionary`. Каждый из них имеет обобщенную и необобщенную версии; тем не менее, необобщенные версии существуют преимущественно для поддержки унаследованного кода.

Иерархия наследования для этих интерфейсов была показана на рис. 7.1. Резюмировать их проще всего следующим образом.

**`IEnumerable<T>` (и `IEnumerable`)**

Предоставляют минимальный уровень функциональности (только перечисление).

**`ICollection<T>` (и `ICollection`)**

Предоставляют средний уровень функциональности (например, свойство `Count`).

**`IList<T>`/`IDictionary<K,V>` и их необобщенные версии**

Предоставляют максимальный уровень функциональности (включая "произвольный" доступ по индексу/ключу).

Потребность в *реализации* любого из упомянутых интерфейсов возникает редко. Когда необходимо написать класс коллекции, почти во всех случаях можно создавать подкласс класса `Collection<T>` (см. раздел "Настраиваемые коллекции и прокси" далее в главе). Язык LINQ предлагает еще один вариант, охватывающий множество сценариев.

Обобщенная и необобщенная версии отличаются между собой сильнее, чем можно было бы ожидать, особенно в случае интерфейса `ICollection`. Причины по большей части исторические: поскольку обобщения появились позже, обобщенные интерфейсы были разработаны с оглядкой на прошлое, что привело к отличающемуся (и лучшему) подбору членов. В итоге интерфейс `ICollection<T>` не расширяет `ICollection`, `IList<T>` не расширяет `IList`, а `IDictionary<TKey, TValue>` не расширяет `IDictionary`. Разумеется, сам класс коллекции свободен в реализации обеих версий интерфейса, если это приносит пользу (часто именно так и есть).

Другая, более тонкая причина того, что интерфейс `IList<T>` не расширяет `IList`, заключается в том, что тогда приведение к `IList<T>` возвращало бы реализацию интерфейса с членами `Add(T)` и `Add(object)`. В результате нарушилась бы статическая безопасность типов, потому что метод `Add` можно было бы вызывать с объектом любого типа.

В текущем разделе рассматриваются интерфейсы `ICollection<T>`, `IList<T>` и их необобщенные версии, а в разделе "Словари" далее в главе обсуждаются интерфейсы словарей.

Не существует *разумного* объяснения способа применения слов *коллекция* и *список* в рамках всей инфраструктуры .NET Framework. Например, так как `IList<T>` является более функциональной версией `ICollection<T>`, можно было бы ожидать, что класс `List<T>` должен быть соответственно более функциональным, чем класс `Collection<T>`. Но это не так. Лучше всего считать термины *коллекция* и *список* в широком смысле синонимами кроме случаев, когда речь идет о конкретном типе.

## ICollection<T> и ICollection

ICollection<T> – стандартный интерфейс для коллекций объектов с поддержкой подсчета. Он предоставляет возможность определения размера коллекции (Count), выяснения, содержится ли элемент в коллекции (Contains), копирования коллекции в массив (ToArray) и определения, предназначена ли коллекция только для чтения (IsReadOnly). Для записываемых коллекций можно также добавлять (Add), удалять (Remove) и очищать (Clear) элементы коллекции. Из-за того, что интерфейс ICollection<T> расширяет IEnumerable<T>, можно также совершать обход с помощью оператора foreach:

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }
  bool Contains (T item);
  void CopyTo (T[] array, int arrayIndex);
  bool IsReadOnly { get; }
  void Add(T item);
  bool Remove (T item);
  void Clear();
}
```

Необобщенный интерфейс ICollection похож в том, что предоставляет коллекцию с возможностью подсчета, но не предлагает функциональности для изменения списка или проверки членства элементов:

```
public interface ICollection : IEnumerable
{
   int Count { get; }
   bool IsSynchronized { get; }
   object SyncRoot { get; }
   void CopyTo (Array array, int index);
}
```

В необобщенном интерфейсе также определены свойства для содействия в синхронизации (глава 14) – они были помещены в обобщенную версию, потому что безопасность потоков больше не считается внутренне присущей коллекции.

Оба интерфейса довольно прямолинейны в реализации. Если реализуется интерфейс ICollection<T>, допускающий только чтение, то методы Add, Remove и Clear должны генерировать исключение NotSupportedException.

Эти интерфейсы обычно реализуются в сочетании с интерфейсом IList или IDictionary.

## IList<T> и IList

IList<T> – стандартный интерфейс для коллекций, поддерживающих индексацию по позиции. В дополнение к функциональности, унаследованной от интерфейсов ICollection<T> и IEnumerable<T>, он предлагает возможность чтения или записи элемента по позиции (через индексатор) и вставки/удаления по позиции:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  T this [int index] { get; set; }
  int IndexOf (T item);
  void Insert (int index, T item);
  void RemoveAt (int index);
}
```

Методы IndexOf выполняют линейный поиск в списке, возвращая -1, если указанный элемент не найден.

Необобщенная версия IList имеет большее число членов, т.к. она наследует меньшее их количество от ICollection:

```
public interface IList : ICollection, IEnumerable
{
  object this [int index] { get; set }
  bool IsFixedSize { get; }
  bool IsReadOnly  { get; }
  int  Add      (object value);
  void Clear();
  bool Contains (object value);
  int  IndexOf  (object value);
  void Insert   (int index, object value);
  void Remove   (object value);
  void RemoveAt (int index);
}
```

Метод Add необобщенного интерфейса IList возвращает целочисленное значение — индекс вновь добавленного элемента. Напротив, метод Add интерфейса ICollection<T> имеет возвращаемый тип void.

Универсальный класс List<T> является наиболее типичной реализацией обоих интерфейсов IList<T> и IList. Массивы в C# также реализуют обобщенную и необобщенную версии IList (хотя методы добавления или удаления элементов скрыты через явную реализацию интерфейса и в случае вызова генерируют исключение NotSupportedException).

При попытке доступа в многомерный массив через индексатор IList генерируется исключение ArgumentException. Такая опасность возникает при написании методов вроде показанного ниже:

```
public object FirstOrNull (IList list)
{
  if (list == null || list.Count == 0) return null;
  return list[0];
}
```

Код может выглядеть "пуленепробиваемым", однако он приведет к генерации исключения в случае вызова с многомерным массивом. Проверить во время выполнения, является ли массив многомерным, можно с помощью следующего кода (за дополнительными сведениями обращайтесь в главу 19):

```
list.GetType().IsArray && list.GetType().GetArrayRank()>1
```

## IReadOnlyList<T>

В версии .NET Framework 4.5 появился новый интерфейс коллекций по имени IReadOnlyList<T> для взаимодействия с коллекциями Windows Runtime, допускающими только чтение. Он полезен сам по себе и может рассматриваться как усеченная версия IList<T>, открывающая доступ лишь к членам, которые требуются для выполнения операций только для чтения на списках:

```
public interface IReadOnlyList<out T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }
  T this[int index] { get; }
}
```

Из-за использования параметра типа данного интерфейса только в выходных позициях он помечается как ковариантный, что позволяет трактовать список кошек, например, как список животных, предназначенный только для чтения. Напротив, тип T не помечается как ковариантный в IList<T>, т.к. T присутствует и во входных, и в выходных позициях.

Интерфейс IReadOnlyList<T> является *представлением* списка, предназначенным только для чтения. Это не обязательно означает, что лежащая в основе реализация допускает только чтение.

Было бы вполне логично, чтобы интерфейс IList<T> порождался от IReadOnlyList<T>. Тем не менее, в Microsoft не смогли внести такое изменение, поскольку оно потребовало бы перемещения членов из IList<T> в IReadOnlyList<T>, что ввело бы критическое изменение в среду CLR 4.5 (во избежание ошибок времени выполнения потребителям пришлось бы перекомпилировать свои программы). Взамен необходимо вручную добавлять IReadOnlyList<T> в списки реализованных интерфейсов для классов, реализующих IList<T>.

Интерфейс IReadOnlyList<T> отображается на тип Windows Runtime по имени IVectorView<T>.

# Класс Array

Класс Array представляет собой неявный базовый класс для всех одномерных и многомерных массивов, а также один из наиболее фундаментальных типов, реализующих стандартные интерфейсы коллекций. Класс Array обеспечивает унификацию типов, так что общий набор методов доступен всем массивам независимо от их объявления или типа элементов.

По причине настолько фундаментального характера массивов язык C# предлагает явный синтаксис для их объявления и инициализации, который был описан в главах 2 и 3. Когда массив объявляется с применением синтаксиса C#, среда CLR неявно создает подтип класса Array за счет синтеза *псевдотипа*, подходящего для размерностей и типов элементов массива. Псевдотип реализует типизированные необобщенные интерфейсы коллекций вроде IList<string>.

Среда CLR также трактует типы массивов специальным образом при конструировании, выделяя им непрерывный участок в памяти. Это делает индексацию в массивах высокоэффективной, но препятствует изменению их размеров в будущем.

Класс Array реализует интерфейсы коллекций вплоть до IList<T> в обеих формах — обобщенной и необобщенной. Однако сам интерфейс IList<T> реализован явно, чтобы сохранить открытый интерфейс Array свободным от методов, подобных Add или Remove, которые генерируют исключение на коллекциях фиксированной длины, таких как массивы. Класс Array в действительности предлагает статический метод Resize, хотя он работает путем создания нового массива и копирования в него всех элементов. Вдобавок к такой неэффективности ссылки на массив в других местах программы будут по-прежнему указывать на его исходную версию. Более удачное решение для коллекций с изменяемыми размерами предусматривает использование класса List<T> (описанного в следующем разделе).

Массив может содержать элементы, имеющие тип значения или ссылочный тип. Элементы типа значения хранятся в массиве на месте, а потому массив из трех элементов типа long (по 8 байтов каждое) будет занимать непрерывный участок памяти раз-

мером 24 байта. С другой стороны, элементы ссылочного типа занимают только объем, требующийся для ссылки (4 байта в 32-разрядной среде или 8 байтов в 64-разрядной среде). На рис. 7.2 показано распределение в памяти для приведенного ниже кода:

```
StringBuilder[] builders = new StringBuilder [5];
builders [0] = new StringBuilder ("builder1");
builders [1] = new StringBuilder ("builder2");
builders [2] = new StringBuilder ("builder3");

long[] numbers = new long [3];
numbers [0] = 12345;
numbers [1] = 54321;
```

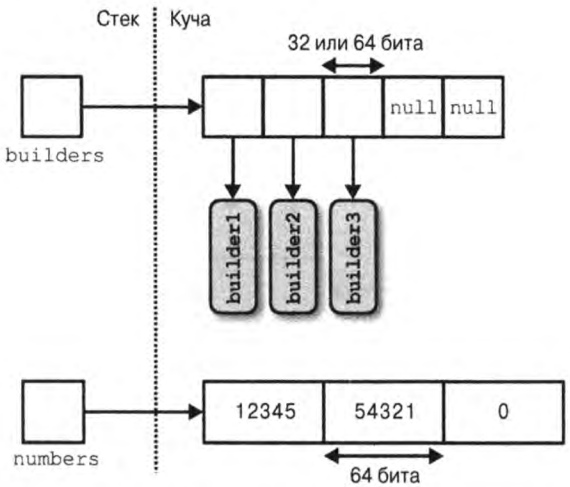



***Рис. 7.2.** Массивы в памяти*

Из-за того, что `Array` представляет собой класс, массивы всегда (сами по себе) являются ссылочными типами независимо от типа элементов массива. Это значит, что оператор `arrayB = arrayA` даст в результате две переменные, которые ссылаются на один и тот же массив. Аналогично два разных массива всегда будут приводить к отрицательному результату при проверке эквивалентности — если только не применяется специальный компаратор эквивалентности. В версии .NET Framework 4.0 появился компаратор для сравнения элементов в массивах или кортежах, доступ к которому можно получить через тип `StructuralComparisons`:

```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };

Console.WriteLine (a1 == a2);          // False
Console.WriteLine (a1.Equals (a2));    // False

IStructuralEquatable se1 = a1;
Console.WriteLine (se1.Equals (a2,
  StructuralComparisons.StructuralEqualityComparer));  // True
```

Массивы можно дублировать с помощью метода `Clone`: `arrayB = arrayA.Clone()`. Тем не менее, результатом будет поверхностная (неглубокая) копия, означающая копирование только участка памяти, в котором представлен собственно массив. Если массив содержит объекты типов значений, тогда копируются сами значения; если же

массив содержит объекты ссылочных типов, то копируются только ссылки (в итоге давая два массива с членами, которые ссылаются на одни и те же объекты).

На рис. 7.3 показан эффект от добавления в пример следующего кода:

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone();
```

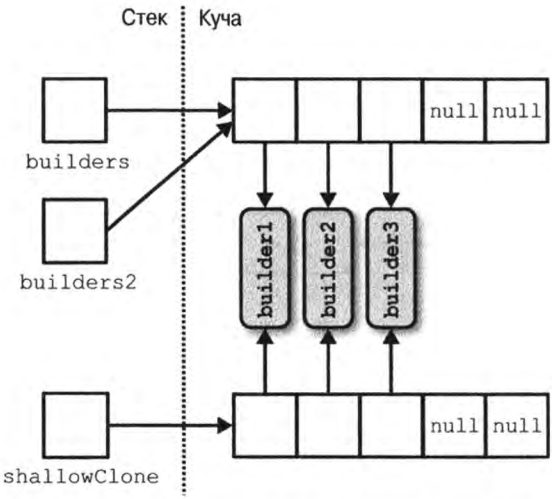



***Рис. 7.3. Поверхностное копирование массива***

Чтобы создать глубокую копию, при которой объекты ссылочных типов дублируются, потребуется пройти в цикле по массиву и клонировать каждый его элемент вручную. Те же самые правила применимы к другим типам коллекций .NET.

Хотя класс `Array` предназначен в первую очередь для использования с 32-битными индексаторами, он также располагает ограниченной поддержкой 64-битных индексаторов (позволяя массиву теоретически адресовать до $2^{64}$ элементов) через несколько методов, которые принимают параметры типов `Int32` и `Int64`. На практике эти перегруженные версии бесполезны, т.к. CLR не разрешает ни одному объекту – включая массивы – занимать более 2 Гбайт (как в 32-разрядной, так и в 64-разрядной среде).

Многие методы в классе `Array`, которые вы, возможно, ожидали видеть как методы экземпляра, на самом деле являются статическими методами. Такое довольно странное проектное решение означает, что при поиске подходящего метода в `Array` следует просматривать и статические методы, и методы экземпляра.

## Конструирование и индексация

Простейший способ создания и индексации массивов предусматривает применение языковых конструкций C#:

```
int[] myArray = { 1, 2, 3 };
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

В качестве альтернативы можно создать экземпляр массива динамически, вызвав метод `Array.CreateInstance`. Это позволяет указывать тип элементов и ранг (количество измерений) во время выполнения – равно как и создавать массивы с индексацией, начинающейся не с нуля, задавая нижнюю границу. Массивы с индексацией, начинающейся не с нуля, не совместимы с общеязыковой спецификацией (Common Language Specification – CLS).

Статические методы `GetValue` и `SetValue` позволяют получать доступ к элементам в динамически созданном массиве (они также работают с обычными массивами):

```
// Создать строковый массив длиной 2 элемента:
Array a = Array.CreateInstance (typeof(string), 2);
a.SetValue ("hi", 0);                  // → a[0] = "hi";
a.SetValue ("there", 1);               // → a[1] = "there";
string s = (string) a.GetValue (0);    // → s = a[0];

// Можно также выполнить приведение к массиву C#:
string[] cSharpArray = (string[]) a;
string s2 = cSharpArray [0];
```

Созданные динамическим образом массивы с индексацией, начинающейся с нуля, могут быть приведены к массиву C# совпадающего или совместимого типа (совместимого согласно стандартным правилам вариантности массивов). Например, если `Apple` является подклассом `Fruit`, то массив `Apple[]` может быть приведен к `Fruit[]`. В результате возникает вопрос о том, почему в качестве унифицированного типа массива вместо класса `Array` не был выбран `object[]`? Дело в том, что массив `object[]` несовместим с многомерными массивами и массивами типов значений (и массивами с индексацией, начинающейся не с нуля). Массив `int[]` не может быть приведен к `object[]`. Таким образом, для полной унификации типов нам требуется класс `Array`.

Методы `GetValue` и `SetValue` также работают с массивами, созданными компилятором, и они полезны при написании методов, которые имеют дело с массивом любого типа и ранга. Для многомерных массивов они принимают *массив* индексаторов:

```
public object GetValue (params int[] indices)
public void   SetValue (object value, params int[] indices)
```

Следующий метод выводит на экран первый элемент любого массива независимо от ранга (количества измерений):

```
void WriteFirstValue (Array a)
{
  Console.Write (a.Rank + "-dimensional; ");

  // Массив индексаторов будет автоматически инициализироваться всеми нулями,
  // поэтому его передача в метод GetValue или SetValue будет приводить к получению
  // или установке основанного на нуле (т.е. первого) элемента в массиве.

  int[] indexers = new int[a.Rank];
  Console.WriteLine ("First value is " +  a.GetValue (indexers));
}

void Demo()
{
  int[]  oneD = { 1, 2, 3 };
  int[,] twoD = { {5,6}, {8,9} };

  WriteFirstValue (oneD);  // одномерный; первое значение равно 1
  WriteFirstValue (twoD);  // двумерный; первое значение равно 5
}
```

Для работы с массивами неизвестного типа, но с известным рангом обобщения предлагают более простое и эффективное решение:

```
void WriteFirstValue<T> (T[] array)
{
  Console.WriteLine (array[0]);
}
```

Метод SetValue генерирует исключение, если элемент имеет тип, несовместимый с массивом. Когда экземпляр массива создан, либо посредством языкового синтаксиса, либо с помощью Array.CreateInstance, его элементы автоматически инициализируются. Для массивов с элементами ссылочных типов это означает занесение в них null, а для массивов с элементами типов значений — вызов стандартного конструктора типа значения (фактически "обнуляющего" члены). Класс Array предоставляет такую функциональность и по запросу через метод Clear:

```
public static void Clear (Array array, int index, int length);
```

Данный метод не влияет на размер массива, что отличается от обычного использования метода Clear (такого как в ICollection<T>.Clear), при котором коллекция сокращается до нуля элементов.

## Перечисление

С массивами легко выполнять перечисление с помощью оператора foreach:

```
int[] myArray = { 1, 2, 3};
foreach (int val in myArray)
  Console.WriteLine (val);
```

Перечисление можно также проводить с применением метода Array.ForEach, определенного следующим образом:

```
public static void ForEach<T> (T[] array, Action<T> action);
```

Здесь используется делегат Action с приведенной ниже сигнатурой:

```
public delegate void Action<T> (T obj);
```

А вот как выглядит первый пример, переписанный с применением метода Array.ForEach:

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

## Длина и ранг

Класс Array предоставляет следующие методы и свойства для запрашивания длины и ранга:

```
public int  GetLength     (int dimension);
public long GetLongLength (int dimension);

public int  Length      { get; }
public long LongLength  { get; }

public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);

public int Rank { get; }  // Возвращает количество измерений в массиве (ранг)
```

Методы GetLength и GetLongLength возвращают длину для заданного измерения (0 для одномерного массива), а свойства Length и LongLength — количество элементов в массиве (включая все измерения).

Методы GetLowerBound и GetUpperBound удобны при работе с массивами, индексы которых начинаются не с нуля. Метод GetUpperBound возвращает такой же результат, как и сложение значений GetLowerBound с GetLength для любого заданного измерения.

## Поиск

Класс Array предлагает набор методов для нахождения элементов внутри одномерного массива.

**Методы BinarySearch**

Для быстрого поиска заданного элемента в отсортированном массиве.

**Методы IndexOf / LastIndex**

Для поиска заданного элемента в несортированном массиве.

**Методы Find/FindLast/FindIndex/FindLastIndex/FindAll/Exists/TrueForAll**

Для поиска элемента (элементов), удовлетворяющих заданному предикату (Predicate<T>), в несортированном массиве.

Ни один из методов поиска в массиве не генерирует исключение в случае, если указанное значение не найдено. Взамен, когда элемент не найден, методы, возвращающие целочисленное значение, возвращают -1 (предполагая, что индекс массива начинается с нуля), а методы, возвращающие обобщенный тип, возвращают стандартное значение для этого типа (скажем, 0 для int или null для string).

Методы двоичного поиска характеризуются высокой скоростью, но работают только с отсортированными массивами и требуют, чтобы элементы сравнивались на предмет *порядка*, а не просто *эквивалентности*. С такой целью методы двоичного поиска могут принимать объект, реализующий интерфейс IComparer или IComparer<T>, который позволяет принимать решения по упорядочению (см. раздел "Подключение протоколов эквивалентности и порядка" далее в главе). Он должен быть согласован с любым компаратором, используемым при исходной сортировке массива. Если компаратор не предоставляется, то будет применен стандартный алгоритм упорядочения типа, основанный на его реализации IComparable/IComparable<T>.

Методы IndexOf и LastIndexOf выполняют простое перечисление по массиву, возвращая первый (или последний) элемент, который совпадает с заданным значением.

Методы поиска на основе предикатов позволяют управлять совпадением заданного элемента с помощью метода делегата или лямбда-выражения. Предикат представляет собой просто делегат, принимающий объект и возвращающий true или false:

```
public delegate bool Predicate<T> (T object);
```

В следующем примере выполняется поиск в массиве строк имени, содержащего букву "a":

```
static void Main()
{
  string[] names = { "Rodney", "Jack", "Jill" };
  string match = Array.Find (names, ContainsA);
  Console.WriteLine (match);  // Jack
}
static bool ContainsA (string name) { return name.Contains ("a"); }
```

Ниже показан тот же пример, сокращенный с помощью анонимного метода:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, delegate (string name)
  { return name.Contains ("a"); } );
```

Лямбда-выражение делает код еще более компактным:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, n => n.Contains ("a")); // Jack
```

Метод `FindAll` возвращает массив всех элементов, удовлетворяющих предикату. На самом деле он эквивалентен методу `Enumerable.Where` из пространства имен `System.Linq` за исключением того, что `FindAll` возвращает массив совпадающих элементов, а не перечисление `IEnumerable<T>` таких элементов.

Метод `Exists` возвращает `true`, если член массива удовлетворяет заданному предикату, и он эквивалентен методу `Any` класса `System.Linq.Enumerable`.

Метод `TrueForAll` возвращает `true`, если все элементы удовлетворяют заданному предикату, и он эквивалентен методу `All` класса `System.Linq.Enumerable`.

## Сортировка

Класс `Array` имеет следующие встроенные методы сортировки:

```
// Для сортировки одиночного массива:

public static void Sort<T> (T[] array);
public static void Sort    (Array array);

// Для сортировки пары массивов:

public static void Sort<TKey,TValue> (TKey[] keys, TValue[] items);
public static void Sort              (Array keys, Array items);
```

Каждый из указанных методов дополнительно перегружен, чтобы также принимать описанные далее аргументы:

```
int index       // Начальный индекс, с которого должна стартовать сортировка
int length              // Количество элементов, подлежащих сортировке
IComparer<T> comparer   // Объект, принимающий решения по упорядочению
Comparison<T> comparison // Делегат, принимающий решения по упорядочению
```

Ниже демонстрируется простейший случай использования метода `Sort`:

```
int[] numbers = { 3, 2, 1 };
Array.Sort (numbers);    // Массив теперь содержит { 1, 2, 3 }
```

Методы, принимающие пару массивов, работают путем переупорядочения элементов каждого массива в тандеме, основываясь на решениях по упорядочению из первого массива. В следующем примере числа и соответствующие им слова сортируются в числовом порядке:

```
int[] numbers = { 3, 2, 1 };
string[] words = { "three", "two", "one" };
Array.Sort (numbers, words);

// Массив numbers теперь содержит { 1, 2, 3 }
// Массив words теперь содержит { "one", "two", "three" }
```

Метод `Array.Sort` требует, чтобы элементы в массиве реализовывали интерфейс `IComparable` (см. раздел "Сравнение порядка" в главе 6). Это означает возможность сортировки для большинства встроенных типов C# (таких как целочисленные типы из предыдущего примера). Если элементы по своей сути несравнимы или нужно пе-

реопределить стандартное упорядочение, то методу Sort потребуется предоставить собственный поставщик сравнения, который будет сообщать относительные позиции двух элементов. Решить задачу можно двумя способами:

- посредством вспомогательного объекта, который реализует интерфейс IComparer/IComparer<T> (см. раздел "Подключение протоколов эквивалентности и порядка" далее в главе);
- посредством делегата Comparison:

```
public delegate int Comparison<T> (T x, T y);
```

Делегат Comparison следует той же семантике, что и метод IComparer<T>.CompareTo: если x находится перед y, то возвращается отрицательное целое число; если x располагается после y, тогда возвращается положительное целое число; если x и y имеют одну и ту же позицию сортировки, то возвращается 0.

В следующем примере мы сортируем массив целых чисел так, чтобы нечетные числа шли первыми:

```
int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort (numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);
// Массив numbers теперь содержит { 1, 3, 5, 2, 4 }
```

В качестве альтернативы вызову метода Sort можно применять операции OrderBy и ThenBy языка LINQ. В отличие от Array.Sort операции LINQ не изменяют исходный массив, а взамен выдают отсортированный результат в новой последовательности IEnumerable<T>.

## Обращение порядка следования элементов

Приведенные ниже методы класса Array изменяют порядок следования всех или части элементов массива на противоположный:

```
public static void Reverse (Array array);
public static void Reverse (Array array, int index, int length);
```

## Копирование

Класс Array предлагает четыре метода для выполнения поверхностного копирования: Clone, CopyTo, Copy и ConstrainedCopy. Первые два являются методами экземпляра, а последние два — статическими методами.

Метод Clone возвращает полностью новый (поверхностно скопированный) массив. Методы CopyTo и Copy копируют непрерывное подмножество элементов массива. Копирование многомерного прямоугольного массива требует отображения многомерного индекса на линейный индекс. Например, позиция [1, 1] в массиве 3 × 3 представляется индексом 4 на основе следующего вычисления: 1*3 + 1. Исходный и целевой диапазоны могут перекрываться без возникновения проблем.

Метод ConstrainedCopy выполняет *атомарную* операцию: если все запрошенные элементы не могут быть успешно скопированы (например, из-за ошибки, связанной с типом), то производится откат всей операции.

Класс Array также предоставляет метод AsReadOnly, который возвращает оболочку, предотвращающую переустановку элементов.

## Преобразование и изменение размера

Метод `Array.ConvertAll` создает и возвращает новый массив элементов типа `TOutput`, вызывая во время копирования элементов предоставленный делегат `Converter`. Делегат `Converter` определен следующим образом:

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

Приведенный ниже код преобразует массив чисел с плавающей точкой в массив целых чисел:

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll (reals, r => Convert.ToInt32 (r));

// Массив wholes содержит { 1, 2, 2 }
```

Метод `Resize` создает новый массив и копирует в него элементы, возвращая новый массив через ссылочный параметр. Любые ссылки на исходный массив в других объектах остаются неизмененными.

Пространство имен `System.Linq` предлагает дополнительный набор расширяющих методов, которые подходят для преобразования массивов. Такие методы возвращают экземпляр реализации интерфейса `IEnumerable<T>`, который можно преобразовать обратно в массив с помощью метода `ToArray` класса `Enumerable`.

# Списки, очереди, стеки и наборы

Инфраструктура .NET Framework предоставляет базовый набор конкретных классов коллекций, которые реализуют интерфейсы, описанные в настоящей главе. В этом разделе внимание сосредоточено на *списковых* коллекциях (в противоположность *словарным* коллекциям, обсуждаемым в разделе "Словари" далее в главе). Как и в случае с ранее рассмотренными интерфейсами, обычно доступен выбор между обобщенной и необобщенной версиями каждого типа. В плане гибкости и производительности обобщенные классы имеют преимущество, делая свои необобщенные аналоги избыточными и предназначенными только для обеспечения обратной совместимости. Ситуация с интерфейсами коллекций иная: их необобщенные версии все еще иногда полезны.

Из описанных в данном разделе классов наиболее часто используется обобщенный класс `List`.

## `List<T>` и `ArrayList`

Обобщенный класс `List` и необобщенный класс `ArrayList` представляют массив объектов с возможностью динамического изменения размера и относятся к числу самых часто применяемых классов коллекций. Класс `ArrayList` реализует интерфейс `IList`, в то время как класс `List<T>` – интерфейсы `IList` и `IList<T>` (а также `IReadOnlyList<T>` – новую версию, допускающую только чтение). В отличие от массивов все интерфейсы реализованы открытым образом, а методы вроде `Add` и `Remove` открыты и работают совершенно ожидаемым образом.

Классы `List<T>` и `ArrayList` функционируют за счет поддержки внутреннего массива объектов, который заменяется более крупным массивом при достижении предельной емкости. Добавление элементов производится эффективно (потому что в конце обычно есть свободные позиции), но вставка элементов может быть медленной (т.к. все элементы после точки вставки должны быть сдвинуты для освобождения по-

зиции). Как и в случае массивов, поиск будет эффективным, если метод BinarySearch используется со списком, который был отсортирован, но иначе он не эффективен, поскольку каждый элемент должен проверяться индивидуально.

Класс List<T> в несколько раз быстрее класса ArrayList, если T является типом значения, т.к. List<T> избегает накладных расходов, связанных с упаковкой и распаковкой элементов.

Классы List<T> и ArrayList предоставляют конструкторы, которые принимают существующую коллекцию элементов: они копируют каждый элемент из нее в новый экземпляр List<T> или ArrayList:

```
public class List<T> : IList<T>, IReadOnlyList<T>
{
  public List ();
  public List (IEnumerable<T> collection);
  public List (int capacity);

  // Добавление и вставка
  public void Add         (T item);
  public void AddRange    (IEnumerable<T> collection);
  public void Insert      (int index, T item);
  public void InsertRange (int index, IEnumerable<T> collection);

  // Удаление
  public bool Remove      (T item);
  public void RemoveAt    (int index);
  public void RemoveRange (int index, int count);
  public int  RemoveAll   (Predicate<T> match);

  // Индексация
  public T this [int index] { get; set; }
  public List<T> GetRange (int index, int count);
  public Enumerator<T> GetEnumerator();

  // Экспортирование, копирование и преобразование
  public T[] ToArray();
  public void CopyTo (T[] array);
  public void CopyTo (T[] array, int arrayIndex);
  public void CopyTo (int index, T[] array, int arrayIndex, int count);
  public ReadOnlyCollection<T> AsReadOnly();
  public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput>
                                    converter);
  // Другие
  public void Reverse();           // Меняет порядок следования элементов
                                   // в списке на противоположный
  public int Capacity { get;set; } //Инициирует расширение внутреннего массива
  public void TrimExcess();        // Усекает внутренний массив до
                                   // фактического количества элементов
  public void Clear();             // Удаляет все элементы, так что Count=0
}

public delegate TOutput Converter <TInput, TOutput> (TInput input);
```

В дополнение к указанным членам класс List<T> предлагает версии экземпляра для всех методов поиска и сортировки из класса Array.

В показанном ниже коде демонстрируется работа свойств и методов List. Примеры поиска и сортировки приводились в разделе "Класс Array" ранее в главе:

```
List<string> words = new List<string>();  // Новый список типа string

words.Add ("melon");
words.Add ("avocado");
words.AddRange (new[] { "banana", "plum" } );
words.Insert (0, "lemon");                                  // Вставить в начале
words.InsertRange (0, new[] { "peach", "nashi" });          // Вставить в начале

words.Remove ("melon");
words.RemoveAt (3);           // Удалить 4-й элемент
words.RemoveRange (0, 2);     // Удалить первые 2 элемента

// Удалить все строки, начинающиеся с n:
words.RemoveAll (s => s.StartsWith ("n"));

Console.WriteLine (words [0]);                              // первое слово
Console.WriteLine (words [words.Count - 1]);                // последнее слово
foreach (string s in words) Console.WriteLine (s);          // все слова
List<string> subset = words.GetRange (1, 2);                // слова со 2-го по 3-е

string[] wordsArray = words.ToArray(); // Создает новый типизированный массив

// Копировать первые два элемента в конец существующего массива:
string[] existing = new string [1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCastWords = words.ConvertAll (s => s.ToUpper());
List<int> lengths = words.ConvertAll (s => s.Length);
```

Необобщенный класс `ArrayList` применяется главным образом для обратной совместимости с кодом .NET Framework 1.x и требует довольно неуклюжих приведений:

```
ArrayList al = new ArrayList();
al.Add ("hello");
string first = (string) al [0];
string[] strArr = (string[]) al.ToArray (typeof (string));
```

Такие приведения не могут быть проверены компилятором; скажем, представленный ниже код скомпилируется успешно, но потерпит неудачу во время выполнения:

```
int first = (int) al [0];  // Исключение во время выполнения
```

Класс `ArrayList` функционально похож на класс `List<object>`. Оба класса удобны, когда необходим список элементов смешанных типов, которые не разделяют общий базовый тип (кроме `object`). В таком случае выбор `ArrayList` может обеспечить преимущество, если в отношении списка должна использоваться рефлексия (глава 19). Рефлексию реализовать проще с помощью необобщенного класса `ArrayList`, чем посредством `List<object>`.

Импортировав пространство имен `System.Linq`, экземпляр `ArrayList` можно преобразовать в обобщенный `List` за счет вызова метода `Cast` и затем `ToList`:

```
ArrayList al = new ArrayList();
al.AddRange (new[] { 1, 5, 9 } );
List<int> list = al.Cast<int>().ToList();
```

`Cast` и `ToList` — расширяющие методы в классе `System.Linq.Enumerable`.

## LinkedList<T>

Класс LinkedList<T> представляет обобщенный двусвязный список (рис. 7.4). Двусвязный список – это цепочка узлов, в которой каждый узел ссылается на предыдущий узел, следующий узел и действительный элемент. Его главное преимущество заключается в том, что элемент может быть эффективно вставлен в любое место списка, т.к. подобное действие требует только создания нового узла и обновления нескольких ссылок. Однако поиск позиции вставки может оказаться медленным ввиду отсутствия в связном списке встроенного механизма для прямой индексации; должен производиться обход каждого узла, и двоичный поиск невозможен.

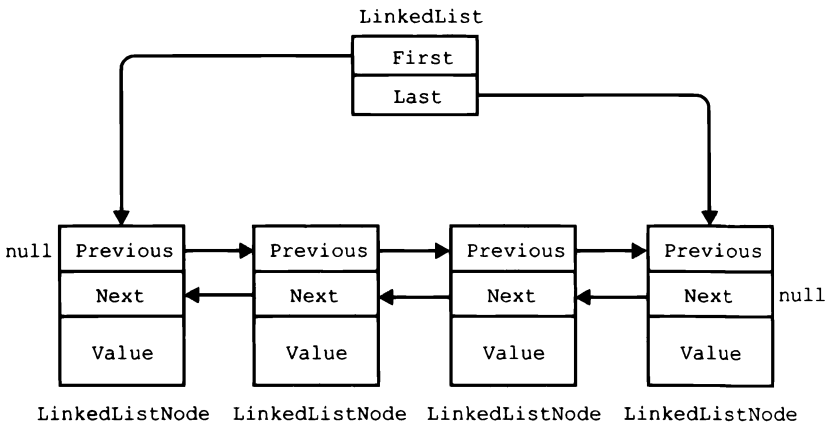



*Рис. 7.4. Класс LinkedList<T>*

Класс LinkedList<T> реализует интерфейсы IEnumerable<T> и ICollection<T> (а также их необобщенные версии), но не IList<T>, поскольку доступ по индексу не поддерживается. Узлы списка реализованы с помощью такого класса:

```
public sealed class LinkedListNode<T>
{
  public LinkedList<T> List { get; }
  public LinkedListNode<T> Next { get; }
  public LinkedListNode<T> Previous { get; }
  public T Value { get; set; }
}
```

При добавлении узла можно указывать его позицию либо относительно другого узла, либо относительно начала/конца списка. Класс LinkedList<T> предоставляет для этого следующие методы:

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst (T value);

public void AddLast (LinkedListNode<T> node);
public LinkedListNode<T> AddLast (T value);

public void AddAfter (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value);

public void AddBefore (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore (LinkedListNode<T> node, T value);
```

Для удаления элементов предлагаются похожие методы:

```
public void Clear();

public void RemoveFirst();
public void RemoveLast();

public bool Remove (T value);
public void Remove (LinkedListNode<T> node);
```

Класс `LinkedList<T>` имеет внутренние поля для отслеживания количества элементов в списке, а также для представления головы и хвоста списка. Доступ к таким полям обеспечивается с помощью следующих открытых свойств:

```
public int Count { get; }                    // Быстрое
public LinkedListNode<T> First { get; }      // Быстрое
public LinkedListNode<T> Last { get; }       // Быстрое
```

Класс `LinkedList<T>` также поддерживает показанные далее методы поиска (каждый из них требует, чтобы список был внутренне перечислимым):

```
public bool Contains (T value);
public LinkedListNode<T> Find (T value);
public LinkedListNode<T> FindLast (T value);
```

Наконец, класс `LinkedList<T>` поддерживает копирование в массив для индексированной обработки и получение перечислителя для работы оператора `foreach`:

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator();
```

Ниже демонстрируется применение класса `LinkedList<string>`:

```
var tune = new LinkedList<string>();
tune.AddFirst ("do");                  // do
tune.AddLast ("so");                   // do - so

tune.AddAfter (tune.First, "re");      // do - re - so
tune.AddAfter (tune.First.Next, "mi"); // do - re - mi - so
tune.AddBefore (tune.Last, "fa");      // do - re - mi - fa - so

tune.RemoveFirst();                    // re - mi - fa - so
tune.RemoveLast();                     // re - mi - fa

LinkedListNode<string> miNode = tune.Find ("mi");
tune.Remove (miNode);                  // re - fa
tune.AddFirst (miNode);                // mi - re - fa

foreach (string s in tune) Console.WriteLine (s);
```

## Queue<T> и Queue

Типы `Queue<T>` и `Queue` — структуры данных FIFO (first-in first-out — первым зашел, первым обслужен; т.е. очередь), предоставляющие методы `Enqueue` (добавление элемента в конец очереди) и `Dequeue` (извлечение и удаление элемента с начала очереди). Также имеется метод `Peek`, предназначенный для возвращения элемента с начала очереди без его удаления, и свойство `Count` (удобно для проверки, существуют ли элементы в очереди, перед их извлечением).

Хотя очереди являются перечислимыми, они не реализуют интерфейс `IList<T>`/`IList`, потому что доступ к членам напрямую по индексу никогда не производится. Тем не менее, есть метод `ToArray`, который предназначен для копирования элементов в массив, где к ним возможен произвольный доступ:

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Queue();
  public Queue (IEnumerable<T> collection); // Копирует существующие элементы
  public Queue (int capacity);     // Сокращает количество изменений размера
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public T Dequeue();
  public void Enqueue (T item);
  public Enumerator<T> GetEnumerator();   // Для поддержки оператора foreach
  public T Peek();
  public T[] ToArray();
  public void TrimExcess();
}
```

Вот пример использования класса `Queue<int>`:

```
var q = new Queue<int>();
q.Enqueue (10);
q.Enqueue (20);
int[] data = q.ToArray();          // Экспортирует в массив
Console.WriteLine (q.Count);       // "2"
Console.WriteLine (q.Peek());      // "10"
Console.WriteLine (q.Dequeue());   // "10"
Console.WriteLine (q.Dequeue());   // "20"
Console.WriteLine (q.Dequeue());   // Генерируется исключение (очередь пуста)
```

Внутренне очереди реализованы с применением массива, который при необходимости расширяется, что очень похоже на обобщенный класс `List`. Очередь поддерживает индексы, которые указывают непосредственно на начальный и хвостовой элементы; таким образом, постановка и извлечение из очереди являются очень быстрыми операциями (кроме случая, когда требуется внутреннее изменение размера).

## Stack<T> и Stack

Типы `Stack<T>` и `Stack` — структуры данных LIFO (last-in first-out — последним зашел, первым обслужен; т.е. стек), которые предоставляют методы `Push` (добавление элемента на верхушку стека) и `Pop` (извлечение и удаление элемента из верхушки стека). Также определены недеструктивный метод `Peek`, свойство `Count` и метод `ToArray` для экспорта данных в массив с целью произвольного доступа к ним:

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Stack();
  public Stack (IEnumerable<T> collection); // Копирует существующие элементы
  public Stack (int capacity);     // Сокращает количество изменений размера
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public Enumerator<T> GetEnumerator();   // Для поддержки оператора foreach
  public T Peek();
  public T Pop();
  public void Push (T item);
  public T[] ToArray();
  public void TrimExcess();
}
```

В следующем примере демонстрируется использование класса Stack<int>:

```
var s = new Stack<int>();
s.Push (1);                 //          Содержимое стека: 1
s.Push (2);                 //          Содержимое стека: 1,2
s.Push (3);                 //          Содержимое стека: 1,2,3
Console.WriteLine (s.Count);  // Выводит 3
Console.WriteLine (s.Peek()); // Выводит 3,  содержимое стека: 1,2,3
Console.WriteLine (s.Pop());  // Выводит 3,  содержимое стека: 1,2
Console.WriteLine (s.Pop());  // Выводит 2,  содержимое стека: 1
Console.WriteLine (s.Pop());  // Выводит 1,  содержимое стека: <пуст>
Console.WriteLine (s.Pop());  // Генерируется исключение
```

Внутренне стеки реализованы с помощью массива, который при необходимости расширяется, что очень похоже на Queue<T> и List<T>.

## BitArray

Класс BitArray представляет собой коллекцию сжатых значений bool с динамически изменяющимся размером. С точки зрения затрат памяти класс BitArray более эффективен, чем простой массив bool и обобщенный список элементов bool, поскольку каждое значение занимает только один бит, в то время как тип bool иначе требует одного байта на значение.

Индексатор BitArray читает и записывает индивидуальные биты:

```
var bits = new BitArray(2);
bits[1] = true;
```

Доступны четыре метода для выполнения побитовых операций (And, Or, Xor и Not). Все они кроме последнего принимают еще один экземпляр BitArray:

```
bits.Xor (bits);  // Побитовое исключающее ИЛИ экземпляра bits с самим собой
Console.WriteLine (bits[1]);  // False
```

## HashSet<T> и SortedSet<T>

Типы HashSet<T> и SortedSet<T> — обобщенные коллекции, которые появились в .NET Framework 3.5 и .NET Framework 4.0 соответственно. Обе они обладают следующим отличительными особенностями:

- их методы Contains выполняются быстро, применяя поиск на основе хеширования;
- они не хранят дублированные элементы и молча игнорируют запросы на добавление дубликатов;
- доступ к элементам по позициям невозможен.

Коллекция SortedSet<T> хранит элементы упорядоченными, тогда как HashSet<T> — нет.

Общность данных типов обеспечивается интерфейсом ISet<T>.

По историческим причинам HashSet<T> находится в сборке *System.Core.dll* (а SortedSet<T> и ISet<T> — в сборке *System.dll*).

Коллекция HashSet<T> реализована с помощью хеш-таблицы, в которой хранятся только ключи, а SortedSet<T> — посредством красно-черного дерева.

Обе коллекции реализуют интерфейс ICollection<T> и предлагают вполне предсказуемые методы, такие как Contains, Add и Remove. Вдобавок предусмотрен метод удаления на основе предиката по имени RemoveWhere.

В следующем коде из существующей коллекции конструируется экземпляр HashSet<char>, затем выполняется проверка членства и перечисление коллекции (обратите внимание на отсутствие дубликатов):

```
var letters = new HashSet<char> ("the quick brown fox");

Console.WriteLine (letters.Contains ('t'));    // true
Console.WriteLine (letters.Contains ('j'));    // false

foreach (char c in letters) Console.Write (c);  // the quickbrownfx
```

(Причина передачи значения string конструктору HashSet<char> объясняется тем, что тип string реализует интерфейс IEnumerable<char>.)

Самый большой интерес вызывают операции над множествами. Приведенные ниже методы операций над множествами являются *деструктивными*, т.к. они модифицируют набор:

```
public void UnionWith           (IEnumerable<T> other);   // Добавляет
public void IntersectWith       (IEnumerable<T> other);   // Удаляет
public void ExceptWith          (IEnumerable<T> other);   // Удаляет
public void SymmetricExceptWith (IEnumerable<T> other);   // Удаляет
```

тогда как следующие методы просто запрашивают набор и потому недеструктивны:

```
public bool IsSubsetOf        (IEnumerable<T> other);
public bool IsProperSubsetOf  (IEnumerable<T> other);
public bool IsSupersetOf      (IEnumerable<T> other);
public bool IsProperSupersetOf(IEnumerable<T> other);
public bool Overlaps          (IEnumerable<T> other);
public bool SetEquals         (IEnumerable<T> other);
```

Метод UnionWith добавляет все элементы из второго набора в исходный набор (исключая дубликаты). Метод IntersectWith удаляет элементы, которые не находятся сразу в обоих наборах. Вот как можно извлечь все гласные из набора символов:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.IntersectWith ("aeiou");
foreach (char c in letters) Console.Write (c);  // euio
```

Метод ExceptWith удаляет указанные элементы из исходного набора. Ниже показано, каким образом удалить все гласные из набора:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.ExceptWith ("aeiou");
foreach (char c in letters) Console.Write (c);  // th qckbrwnfx
```

Метод SymmetricExceptWith удаляет все элементы кроме тех, которые являются уникальными в одном или в другом наборе:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.SymmetricExceptWith ("the lazy brown fox");
foreach (char c in letters) Console.Write (c);  // quicklazy
```

Обратите внимание, что поскольку типы HashSet<T> и SortedSet<T> реализуют интерфейс IEnumerable<T>, в качестве аргумента в любом методе операции над множествами можно использовать другой тип набора (или коллекции).

Тип SortedSet<T> предлагает все члены типа HashSet<T> и вдобавок следующие члены:

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; }
```

Конструктор типа SortedSet<T> также принимает дополнительный параметр типа IComparer<T> (отличающийся от *компаратора эквивалентности*).

Ниже приведен пример загрузки в SortedSet<char> тех же букв, что и ранее:

```
var letters = new SortedSet<char> ("the quick brown fox");
foreach (char c in letters) Console.Write (c);  //  bcefhiknoqrtuwx
```

Исходя из этого, получить буквы между *f* и *j* можно так:

```
foreach (char c in letters.GetViewBetween ('f', 'j'))
  Console.Write (c);  // fhi
```

# Словари

Словарь – это коллекция, в которой каждый элемент является парой "ключ/значение". Словари чаще всего применяются для поиска и представления сортированных списков.

В .NET Framework определен стандартный протокол для словарей через интерфейсы IDictionary и IDictionary<TKey, TValue>, а также набор универсальных классов словарей. Упомянутые классы различаются в следующих отношениях:

- хранятся ли элементы в отсортированной последовательности;
- можно ли получать доступ к элементам по позиции (индексу) и по ключу;
- является ли тип обобщенным или необобщенным;
- является ли тип быстрым или медленным при извлечении элементов по ключу из крупного словаря.

В табл. 7.1 представлена сводка по всем классам словарей, а также описаны отличия в перечисленных выше аспектах. Значения времени указаны в миллисекундах; при тестировании выполнялось 50 000 операций в словаре с целочисленными ключами и значениями на ПК с процессором 1,5 ГГц. (Разница в производительности между обобщенными и необобщенными версиями для одной и той же структуры коллекции объясняется упаковкой и связана только с элементами типов значений.)

С помощью нотации "большое О" время извлечения можно описать следующим образом:

- O(1) для Hashtable, Dictionary и OrderedDictionary;
- O(log *n*) для SortedDictionary и SortedList;
- O(*n*) для ListDictionary (и несловарных типов, таких как List<T>).

где *n* – количество элементов в коллекции.

**Таблица 7.1. Классы словарей**

| Тип | Внутренняя структура | Поддерживается ли извлечение по индексу? | Накладные расходы, связанные с памятью (среднее количество байтов на элемент) | Скорость: произвольная вставка | Скорость: последовательная вставка | Скорость: извлечение по ключу |
|---|---|---|---|---|---|---|
| **Несортированные** | | | | | | |
| `Dictionary<K,V>` | Хеш-таблица | Нет | 22 | 30 | 30 | 20 |
| `Hashtable` | Хеш-таблица | Нет | 38 | 50 | 50 | 30 |
| `ListDictionary` | Связный список | Нет | 36 | 50 000 | 50 000 | 50 000 |
| `OrderedDictionary` | Хеш-таблица + массив | Да | 59 | 70 | 70 | 40 |
| **Сортированные** | | | | | | |
| `SortedDictionary<K,V>` | Красно-черное дерево | Нет | 20 | 130 | 100 | 120 |
| `SortedList<K,V>` | Пара массивов | Да | 2 | 3 300 | 30 | 40 |
| `SortedList` | Пара массивов | Да | 27 | 4 500 | 100 | 180 |

## IDictionary<TKey,TValue>

Интерфейс `IDictionary<TKey,TValue>` определяет стандартный протокол для всех коллекций, основанных на парах "ключ/значение". Он расширяет интерфейс `ICollection<T>`, добавляя методы и свойства для доступа к элементам на основе ключей произвольных типов:

```
public interface IDictionary <TKey, TValue> :
  ICollection <KeyValuePair <TKey, TValue>>, IEnumerable
{
   bool ContainsKey  (TKey key);
   bool TryGetValue  (TKey key, out TValue value);
   void Add          (TKey key, TValue value);
   bool Remove       (TKey key);

   TValue this [TKey key]      { get; set; } // Основной индексатор - по ключу
   ICollection <TKey> Keys     { get; }      // Возвращает только ключи
   ICollection <TValue> Values { get; }      // Возвращает только значения
}
```

Начиная с версии .NET Framework 4.5, есть также интерфейс по имени `IReadOnlyDictionary<TKey,TValue>`, в котором определено подмножество членов словаря, допускающих только чтение. Он отображается на тип `IMapView<K,V>` из Windows Runtime и был введен главным образом по этой причине.

Чтобы добавить элемент в словарь, необходимо либо вызвать метод `Add`, либо воспользоваться средством доступа `set` индекса — в последнем случае элемент добавляет-

ся в словарь, если такой ключ в словаре отсутствует (или производится обновление элемента, если ключ присутствует). Дублированные ключи запрещены во всех реализациях словарей, поэтому вызов метода Add два раза с тем же самым ключом приводит к генерации исключения.

Для извлечения элемента из словаря применяется либо индексатор, либо метод TryGetValue. Если ключ не существует, тогда индексатор генерирует исключение, в то время как метод TryGetValue возвращает false. Можно явно проверить членство, вызвав метод ContainsKey; однако за это придется заплатить двумя поисками, если элемент впоследствии будет извлекаться.

Перечисление прямо по IDictionary<TKey,TValue> возвращает последовательность структур KeyValuePair:

```
public struct KeyValuePair <TKey, TValue>
{
  public TKey Key     { get; }
  public TValue Value { get; }
}
```

С помощью свойств Keys/Values словаря можно выполнять перечисление только по ключам или только по значениям.

Мы продемонстрируем использование интерфейса IDictionary<TKey,TValue> с обобщенным классом Dictionary в следующем разделе.

## IDictionary

Необобщенный интерфейс IDictionary в принципе является таким же, как интерфейс IDictionary<TKey,TValue>, за исключением двух важных функциональных отличий. Эти отличия следует понимать, потому что IDictionary присутствует в унаследованном коде (а местами и в самой инфраструктуре .NET Framework):

- извлечение несуществующего ключа через индексатор дает в результате null (не приводя к генерации исключения);
- членство проверяется с помощью метода Contains, но не ContainsKey.

Перечисление по необобщенному интерфейсу IDictionary возвращает последовательность структур DictionaryEntry:

```
public struct DictionaryEntry
{
  public object Key   { get; set; }
  public object Value { get; set; }
}
```

## Dictionary<TKey,TValue> и Hashtable

Обобщенный класс Dictionary — одна из наиболее часто применяемых коллекций (наряду с коллекцией List<T>). Для хранения ключей и значений он использует структуру данных в форме хеш-таблицы, а также характеризуется высокой скоростью работы и эффективностью.

Необобщенная версия Dictionary<TKey,TValue> называется Hashtable; необобщенного класса, который бы имел имя Dictionary, не существует. Когда мы ссылаемся просто на Dictionary, то имеем в виду обобщенный класс Dictionary<TKey,TValue>.

Класс Dictionary реализует обобщенный и необобщенный интерфейсы IDictionary, и обобщенный интерфейс IDictionary открыт. Фактически Dictionary является "учебной" реализацией обобщенного интерфейса IDictionary.

Ниже показано, как с ним работать:

```
var d = new Dictionary<string, int>();

d.Add("One", 1);
d["Two"] = 2;   // Добавляет в словарь, потому что "two" пока отсутствует
d["Two"] = 22;  // Обновляет словарь, т.к. "two" уже присутствует
d["Three"] = 3;

Console.WriteLine (d["Two"]);               // Выводит "22"
Console.WriteLine (d.ContainsKey ("One")); // true (быстрая операция)
Console.WriteLine (d.ContainsValue (3));   // true (медленная операция)
int val = 0;
if (!d.TryGetValue ("onE", out val))
  Console.WriteLine ("No val");  // "No val" (чувствительно к регистру)

// Три разных способа перечисления словаря:

foreach (KeyValuePair<string, int> kv in d)      //  One; 1
  Console.WriteLine (kv.Key + "; " + kv.Value);  //  Two; 22
                                                 //  Three; 3

foreach (string s in d.Keys) Console.Write (s);  // OneTwoThree
Console.WriteLine();
foreach (int i in d.Values) Console.Write (i);   // 1223
```

Лежащая в основе Dictionary хеш-таблица преобразует ключ каждого элемента в целочисленный хеш-код — псевдоуникальное значение — и затем применяет алгоритм для преобразования хеш-кода в хеш-ключ. Такой хеш-ключ используется внутренне для определения, к какому "сегменту" относится запись. Если сегмент содержит более одного значения, тогда в нем производится линейный поиск. Хорошая хеш-функция не стремится возвращать строго уникальные хеш-коды (что обычно невозможно); она старается вернуть хеш-коды, которые равномерно распределены в пространстве 32-битных целых чисел. Это позволяет избежать сценария с получением нескольких очень крупных (и неэффективных) сегментов.

Благодаря своей возможности определения эквивалентности ключей и получения хеш-кодов словарь может работать с ключами любого типа. По умолчанию эквивалентность определяется с помощью метода object.Equals ключа, а псевдоуникальный хеш-код получается через метод GetHashCode ключа. Такое поведение можно изменить, либо переопределив указанные методы, либо предоставив при конструировании словаря объект, который реализует интерфейс IEqualityComparer. Распространенный случай применения предусматривает указание нечувствительного к регистру компаратора эквивалентности, когда используются строковые ключи:

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

Мы обсудим данный момент более подробно в разделе "Подключение протоколов эквивалентности и порядка" далее в главе.

Как и со многими другими типами коллекций, производительность словаря можно несколько улучшить за счет указания в конструкторе ожидаемого размера коллекции, тем самым избежав или снизив потребность во внутренних операциях изменения размера.

Необобщенная версия имеет имя `Hashtable` и функционально подобна, не считая отличий, которые являются результатом открытия ею необобщенного интерфейса `IDictionary`, как обсуждалось ранее.

Недостаток `Dictionary` и `Hashtable` связан с тем, что элементы не отсортированы. Кроме того, первоначальный порядок, в котором добавлялись элементы, не предохраняется. Как и со всеми словарями, дублированные ключи не разрешены.

Когда в версии .NET Framework 2.0 появились обобщенные коллекции, команда разработчиков CLR решила именовать их согласно тому, что они представляют (`Dictionary`, `List`), а не тому, как они реализованы внутренне (`Hashtable`, `ArrayList`). Хотя такой подход обеспечивает свободу изменения реализации в будущем, он также означает, что в имени больше не отражается *контракт производительности* (зачастую являющийся самым важным критерием при выборе одного вида коллекции из нескольких доступных).

## OrderedDictionary

Класс `OrderedDictionary` — необобщенный словарь, который хранит элементы в порядке их добавления. С помощью `OrderedDictionary` получать доступ к элементам можно и по индексам, и по ключам.

Класс `OrderedDictionary` не является *отсортированным* словарем.

Класс `OrderedDictionary` представляет собой комбинацию классов `Hashtable` и `ArrayList`. Таким образом, он обладает всей функциональностью `Hashtable`, а также имеет функции вроде `RemoveAt` и целочисленный индексатор. Кроме того, класс `OrderedDictionary` открывает доступ к свойствам `Keys` и `Values`, которые возвращают элементы в их исходном порядке.

Класс `OrderedDictionary` появился в .NET Framework 2.0 и, как ни странно, его обобщенная версия отсутствует.

## ListDictionary и HybridDictionary

Для хранения лежащих в основе данных в классе `ListDictionary` применяется односвязный список. Он не обеспечивает сортировку, хотя предохраняет исходный порядок элементов. С большими списками класс `ListDictionary` работает исключительно медленно. Единственным "предметом гордости" можно считать его эффективность в случае очень маленьких списков (менее 10 элементов).

Класс `HybridDictionary` — это `ListDictionary`, который автоматически преобразуется в `Hashtable` при достижении определенного размера, решая проблемы низкой производительности класса `ListDictionary`. Идея в том, чтобы обеспечить низкое потребление памяти для мелких словарей и хорошую производительность для крупных словарей. Однако, учитывая накладные расходы, которые сопровождают переход от одного класса к другому, а также тот факт, что класс `Dictionary` довольно неплох в любом сценарии, вполне разумно использовать `Dictionary` с самого начала.

Оба класса доступны только в необобщенной форме.

## Отсортированные словари

В .NET Framework предлагаются два класса словарей, которые внутренне устроены так, что их содержимое всегда сортируется по ключу:

- `SortedDictionary<TKey,TValue>`
- `SortedList<TKey,TValue>`[1]

(В настоящем разделе мы будем сокращать `<TKey,TValue>` до `<,>`.)

Класс `SortedDictionary<,>` применяет красно-черное дерево — структуру данных, которая спроектирована так, что работает одинаково хорошо в любом сценарии вставки либо извлечения.

Класс `SortedList<,>` внутренне реализован с помощью пары упорядоченных массивов, обеспечивая высокую производительность извлечения (посредством двоичного поиска), но низкую производительность вставки (поскольку существующие значения должны сдвигаться, чтобы освободить место под новый элемент).

Класс `SortedDictionary<,>` намного быстрее класса `SortedList<,>` при вставке элементов в произвольном порядке (особенно в случае крупных списков). Тем не менее, класс `SortedList<,>` обладает дополнительной возможностью: доступом к элементам по индексу, а также по ключу. Благодаря отсортированному списку можно переходить непосредственно к *n*-ному элементу в отсортированной последовательности (с помощью индексатора на свойствах `Keys/Values`). Чтобы сделать то же самое с помощью `SortedDictionary<,>`, потребуется вручную пройти через *n* элементов. (В качестве альтернативы можно было бы написать класс, комбинирующий отсортированный словарь и список.)

Ни одна из трех коллекций не допускает наличие дублированных ключей (как и все словари).

В следующем примере используется рефлексия с целью загрузки всех методов, определенных в классе `System.Object`, внутрь отсортированного списка с ключами по именам, после чего производится перечисление их ключей и значений:

```
// Класс MethodInfo находится в пространстве имен System.Reflection
var sorted = new SortedList <string, MethodInfo>();
foreach (MethodInfo m in typeof (object).GetMethods())
  sorted [m.Name] = m;
foreach (string name in sorted.Keys)
  Console.WriteLine (name);
foreach (MethodInfo m in sorted.Values)
  Console.WriteLine (m.Name + " returns a " + m.ReturnType);
```

Ниже показаны результаты первого перечисления:

```
Equals
GetHashCode
GetType
ReferenceEquals
ToString
```

А вот результаты второго перечисления:

[1] Существует также функционально идентичная ему необобщенная версия по имени `SortedList`.

```
Equals returns a System.Boolean
GetHashCode returns a System.Int32
GetType returns a System.Type
ReferenceEquals returns a System.Boolean
ToString returns a System.String
```

Обратите внимание, что словарь наполняется через свой индексатор. Если взамен применить метод Add, то сгенерируется исключение, т.к. в классе object, на котором осуществляется рефлексия, метод Equals перегружен, и добавить в словарь тот же самый ключ два раза не получится. В случае использования индексатора элемент, добавляемый позже, переписывает элемент, добавленный раньше, предотвращая возникновение такой ошибки.

Можно сохранять несколько членов одного ключа, делая каждый элемент значения списком:

```
SortedList <string, List<MethodInfo>>
```

Расширяя рассматриваемый пример, следующий код извлекает объект MethodInfo с ключом "GetHashCode", точно как в случае обычного словаря:

```
Console.WriteLine (sorted ["GetHashCode"]);  // Int32 GetHashCode()
```

Весь написанный до сих пор код будет также работать с классом SortedDictionary<,>. Однако показанные ниже две строки кода, которые извлекают последний ключ и значение, работают только с отсортированным списком:

```
Console.WriteLine (sorted.Keys  [sorted.Count - 1]);          // ToString
Console.WriteLine (sorted.Values[sorted.Count - 1].IsVirtual); // True
```

# Настраиваемые коллекции и прокси

Классы коллекций, которые обсуждались в предшествующих разделах, удобны своей возможностью непосредственного создания экземпляров, но они не позволяют управлять тем, что происходит, когда элемент добавляется или удаляется из коллекции. Необходимость в таком контроле периодически возникает у строго типизированных коллекций в приложении, например:

- для запуска события, когда элемент добавляется или удаляется;
- для обновления свойств из-за добавления или удаления элемента;
- для обнаружения "несанкционированной" операции добавления/удаления и генерации исключения (скажем, если операция нарушает бизнес-правило).

Именно по таким причинам инфраструктура .NET Framework предлагает классы коллекций, определенные в пространстве имен System.Collections.ObjectModel. По существу они являются прокси, или оболочками, реализующими интерфейс IList<T> либо IDictionary<,>, которые переадресуют вызовы методам внутренней коллекции. Каждая операция Add, Remove или Clear проходит через виртуальный метод, действующий в качестве "шлюза", когда он переопределен.

Классы настраиваемых коллекций обычно применяются для коллекций, открытых публично; например, в классе System.Windows.Form публично открыта коллекция элементов управления.

## Collection<T> и CollectionBase

Класс Collection<T> является настраиваемой оболочкой для List<T>. Помимо реализации интерфейсов IList<T> и IList в нем определены четыре дополнительных виртуальных метода и защищенное свойство:

```
public class Collection<T> :
  IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
   // ...
   protected virtual void ClearItems();
   protected virtual void InsertItem (int index, T item);
   protected virtual void RemoveItem (int index);
   protected virtual void SetItem (int index, T item);

   protected IList<T> Items { get; }
}
```

Виртуальные методы предоставляют шлюз, с помощью которого можно "привязаться" с целью изменения или расширения нормального поведения списка. Защищенное свойство Items позволяет реализующему коду получать прямой доступ во "внутренний список" – это используется для внесения изменений внутренне без запуска виртуальных методов.

Виртуальные методы переопределять не обязательно; их можно оставить незатронутыми, если только не возникает потребность в изменении стандартного поведения списка. В следующем примере показан типичный "скелет" программы, в которой применяется класс Collection<T>:

```
public class Animal
{
  public string Name;
  public int Popularity;

  public Animal (string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}
public class AnimalCollection : Collection <Animal>
{
  // AnimalCollection - уже полностью функционирующий список животных.
  // Никакого дополнительного кода не требуется.
}
public class Zoo     // Класс, который откроет доступ к AnimalCollection.
{                    // Обычно он может иметь дополнительные члены.
  public readonly AnimalCollection Animals = new AnimalCollection();
}

class Program
{
  static void Main()
  {
    Zoo zoo = new Zoo();
    zoo.Animals.Add (new Animal ("Kangaroo", 10));
    zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
    foreach (Animal a in zoo.Animals) Console.WriteLine (a.Name);
  }
}
```

Как можно заметить, класс AnimalCollection не обладает большей функциональностью, чем простой класс List<Animal>; его роль заключается в том, чтобы предоставить базу для будущего расширения. В целях иллюстрации мы добавим к Animal свойство Zoo, так что экземпляр животного может ссылаться на зоопарк (zoo), в котором оно содержится, и переопределим все виртуальные методы в Collection<Animal> для автоматической поддержки этого свойства:

```
public class Animal
{
  public string Name;
  public int Popularity;
  public Zoo Zoo { get; internal set; }
  public Animal(string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : Collection <Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
  protected override void InsertItem (int index, Animal item)
  {
    base.InsertItem (index, item);
    item.Zoo = zoo;
  }
  protected override void SetItem (int index, Animal item)
  {
    base.SetItem (index, item);
    item.Zoo = zoo;
  }
  protected override void RemoveItem (int index)
  {
    this [index].Zoo = null;
    base.RemoveItem (index);
  }
  protected override void ClearItems()
  {
    foreach (Animal a in this) a.Zoo = null;
    base.ClearItems();
  }
}

public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
}
```

Класс Collection<T> также имеет конструктор, который принимает существующую реализацию IList<T>. В отличие от других классов коллекций передаваемый список не *копируется*, а для него *создается прокси*, т.е. последующие изменения будут отражаться в оболочке Collection<T> (хотя и *без* запуска виртуальных методов Collection<T>). И наоборот, изменения, внесенные через Collection<T>, будут воздействовать на лежащий в основе список.

### CollectionBase

Класс CollectionBase – это необобщенная версия Collection<T>, появившаяся в .NET Framework 1.0. Он поддерживает большинство тех же возможностей, что и Collection<T>, но менее удобен в использовании. Вместо шаблонных методов InsertItem, RemoveItem, SetItem и ClearItem класс CollectionBase имеет методы "привязки", что удваивает количество требуемых методов: OnInsert, OnInsertComplete, OnSet, OnSetComplete, OnRemove, OnRemoveComplete, OnClear и OnClearComplete. Поскольку класс CollectionBase не является обобщенным, при создании его подклассов понадобится также реализовать типизированные методы – как минимум, типизированный индексатор и метод Add.

## KeyedCollection<TKey,TItem> и DictionaryBase

Класс KeyedCollection<TKey,TItem> представляет собой подкласс класса Collection<TItem>. Определенная функциональность в него добавлена, а определенная – удалена. К добавленной функциональности относится возможность доступа к элементам по ключу, почти как в словаре. Удаление функциональности касается устранения возможности создавать прокси для собственного внутреннего списка.

Коллекция с ключами имеет некоторое сходство с классом OrderedDictionary в том, что комбинирует линейный список с хеш-таблицей. Тем не менее, в отличие от OrderedDictionary коллекция с ключами не реализует интерфейс IDictionary и не поддерживает концепцию *пары* "ключ/значение". Взамен ключи получаются из самих элементов: через абстрактный метод GetKeyForItem. Это означает, что перечисление по коллекции с ключами производится точно так же, как в обычном списке.

Класс KeyedCollection<TKey,TItem> лучше всего воспринимать как класс Collection<TItem> с добавочным быстрым поиском по ключу.

Поскольку коллекция с ключами является подклассом класса Collection<>, она наследует всю функциональность Collection<> кроме возможности указания существующего списка при конструировании. В классе KeyedCollection<TKey,TItem> определены дополнительные члены, как показано ниже:

```
public abstract class KeyedCollection <TKey, TItem> : Collection <TItem>
  // ...
  protected abstract TKey GetKeyForItem(TItem item);
  protected void ChangeItemKey(TItem item, TKey newKey);
  // Быстрый поиск по ключу - является дополнением к поиску по индексу
  public TItem this[TKey key] { get; }
  protected IDictionary<TKey, TItem> Dictionary { get; }
}
```

Метод GetKeyForItem переопределяется для получения ключа элемента из лежащего в основе объекта. Метод ChangeItemKey должен вызываться, если свойство ключа элемента изменяется, чтобы обновить внутренний словарь. Свойство Dictionary возвращает внутренний словарь, применяемый для реализации поиска, который создается при добавлении первого элемента. Такое поведение можно изменить, указав в конструкторе порог создания, что отсрочит создание внутреннего словаря до момента, когда будет достигнуто пороговое значение (а тем временем при поступлении запроса элемента по ключу будет выполняться линейный поиск). Веская причина, по которой порог создания не указывается, связана с тем, что наличие допустимого словаря может быть полезно в получении коллекции ICollection<> ключей через

свойство Keys класса Dictionary. Затем данная коллекция может быть передана открытому свойству.

Самое распространенное использование класса KeyedCollection<,> связано с предоставлением коллекции элементов, доступных как по индексу, так и по имени. В целях демонстрации мы реализуем класс AnimalCollection в виде KeyedCollection<string,Animal>:

```
public class Animal
{
  string name;
  public string Name
  {
    get { return name; }
    set {
      if (Zoo != null) Zoo.Animals.NotifyNameChange (this, value);
      name = value;
    }
  }
  public int Popularity;
  public Zoo Zoo { get; internal set; }

  public Animal (string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : KeyedCollection <string, Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }

  internal void NotifyNameChange (Animal a, string newName)
  {
    this.ChangeItemKey (a, newName);
  }

  protected override string GetKeyForItem (Animal item)
  {
    return item.Name;
  }

  // Следующие методы должны быть реализованы так же, как в предыдущем примере
  protected override void InsertItem (int index, Animal item)...
  protected override void SetItem (int index, Animal item)...
  protected override void RemoveItem (int index)...
  protected override void ClearItems()...
}

public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
}

class Program
{
  static void Main()
  {
    Zoo zoo = new Zoo();
```

```
    zoo.Animals.Add (new Animal ("Kangaroo", 10));
    zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
    Console.WriteLine (zoo.Animals [0].Popularity);                // 10
    Console.WriteLine (zoo.Animals ["Mr Sea Lion"].Popularity);    // 20
    zoo.Animals ["Kangaroo"].Name = "Mr Roo";
    Console.WriteLine (zoo.Animals ["Mr Roo"].Popularity);         // 10
  }
}
```

### DictionaryBase

Необобщенная версия коллекции KeyedCollection называется DictionaryBase. Этот унаследованный класс существенно отличается принятым в нем подходом: он реализует интерфейс IDictionary и подобно классу CollectionBase применяет множество неуклюжих методов привязки: OnInsert, OnInsertComplete, OnSet, OnSetComplete, OnRemove, OnRemoveComplete, OnClear и OnClearComplete (и дополнительно OnGet). Главное преимущество реализации IDictionary вместо принятия подхода с KeyedCollection состоит в том, что для получения ключей не требуется создавать его подкласс. Но поскольку основным назначением DictionaryBase является создание подклассов, в итоге какие-либо преимущества вообще отсутствуют. Улучшенная модель в KeyedCollection почти наверняка объясняется тем фактом, что данный класс был написан несколькими годами позже, с оглядкой на прошлый опыт. Класс DictionaryBase лучше всего рассматривать как предназначенный для обратной совместимости.

## ReadOnlyCollection<T>

Класс ReadOnlyCollection<T> — это оболочка, или *прокси*, которая является представлением коллекции, доступным только для чтения. Он полезен в ситуации, когда нужно разрешить классу открывать доступ только для чтения к коллекции, которую данный класс может внутренне обновлять.

Конструктор класса ReadOnlyCollection<T> принимает входную коллекцию, на которую он поддерживает постоянную ссылку. Он не создает статическую копию входной коллекции, поэтому последующие изменения входной коллекции будут видны через оболочку, допускающую только чтение.

В целях иллюстрации предположим, что классу необходимо предоставить открытый доступ только для чтения к списку строк по имени Names:

```
public class Test
{
  public List<string> Names { get; private set; }
}
```

Сделана лишь половина работы. Хотя другие типы не могут переустанавливать свойство Names, они по-прежнему могут вызывать методы Add, Remove или Clear на списке. Проблему решает класс ReadOnlyCollection<T>:

```
public class Test
{
  List<string> names;
  public ReadOnlyCollection<string> Names { get; private set; }

  public Test()
  {
```

```
    names = new List<string>();
    Names = new ReadOnlyCollection<string> (names);
  }
  public void AddInternally() { names.Add ("test"); }
}
```

Теперь изменять список имен могут только члены класса `Test`:

```
Test t = new Test();

Console.WriteLine (t.Names.Count);      // 0
t.AddInternally();
Console.WriteLine (t.Names.Count);      // 1

t.Names.Add ("test");                   // Ошибка на этапе компиляции
((IList<string>) t.Names).Add ("test"); // Генерируется исключение
                                        // NotSupportedException
```

# Подключение протоколов эквивалентности и порядка

В разделах "Сравнение эквивалентности" и "Сравнение порядка" главы 6 мы описали стандартные протоколы .NET, которые привносят в тип возможность эквивалентности, хеширования и сравнения. Тип, который реализует упомянутые протоколы, может корректно функционировать в словаре или отсортированном списке. В частности:

- тип, для которого методы `Equals` и `GetHashCode` возвращают осмысленные результаты, может использоваться в качестве ключа в `Dictionary` или `Hashtable`;
- тип, который реализует `IComparable/IComparable<T>`, может применяться в качестве ключа в любом *отсортированном* словаре или списке.

Стандартная реализация эквивалентности или сравнения типа обычно отражает то, что является наиболее "естественным" для данного типа. Тем не менее, иногда стандартное поведение не подходит. Может понадобиться словарь, в котором ключ строкового типа трактуется в нечувствительной к регистру манере. Или же может потребоваться список заказчиков, отсортированный по их почтовым кодам. По этой причине в .NET Framework также определен соответствующий набор "подключаемых" протоколов. Подключаемые протоколы служат двум целям:

- они позволяют переключаться на альтернативное поведение эквивалентности или сравнения;
- они позволяют использовать словарь или отсортированную коллекцию с типом ключа, который не обладает внутренней возможностью эквивалентности или сравнения.

Подключаемые протоколы состоят из указанных ниже интерфейсов.

**`IEqualityComparer` и `IEqualityComparer<T>`**

- Выполняют подключаемое *сравнение эквивалентности и хеширование.*
- Распознаются классами `Hashtable` и `Dictionary`.

**IComparer и IComparer<T>**

- Выполняют подключаемое *сравнение порядка*.
- Распознаются отсортированными словарями и коллекциями, а также методом `Array.Sort`.

Каждый интерфейс доступен в обобщенной и необобщенной формах. Интерфейсы `IEqualityComparer` также имеют стандартную реализацию в классе по имени `EqualityComparer`.

Кроме того, в .NET Framework 4.0 появились два новых интерфейса `IStructuralEquatable` и `IStructuralComparable`, которые позволяют выполнять структурные сравнения для классов и массивов.

## IEqualityComparer и EqualityComparer

Компаратор эквивалентности позволяет переключаться на нестандартное поведение эквивалентности и хеширования главным образом для классов `Dictionary` и `Hashtable`.

Вспомним требования к словарю, основанному на хеш-таблице. Для любого заданного ключа он должен отвечать на два следующих вопроса.

- Является ли указанный ключ таким же, как и другой?
- Какой целочисленный хеш-код имеет указанный ключ?

Компаратор эквивалентности отвечает на такие вопросы путем реализации интерфейсов `IEqualityComparer`:

```
public interface IEqualityComparer<T>
{
  bool Equals (T x, T y);
  int GetHashCode (T obj);
}
public interface IEqualityComparer  // Необобщенная версия
{
  bool Equals (object x, object y);
  int GetHashCode (object obj);
}
```

Для написания специального компаратора необходимо реализовать один или оба интерфейса (реализация обоих интерфейсов обеспечивает максимальную степень взаимодействия). Учитывая, что это несколько утомительно, в качестве альтернативы можно создать подкласс абстрактного класса `EqualityComparer`, определение которого показано ниже:

```
public abstract class EqualityComparer<T> : IEqualityComparer,
                                            IEqualityComparer<T>
{
  public abstract bool Equals (T x, T y);
  public abstract int GetHashCode (T obj);

  bool IEqualityComparer.Equals (object x, object y);
  int IEqualityComparer.GetHashCode (object obj);

  public static EqualityComparer<T> Default { get; }
}
```

Класс EqualityComparer реализует оба интерфейса; ваша работа сводится к тому, чтобы просто переопределить два абстрактных метода.

Семантика методов Equals и GetHashCode подчиняется тем же правилам для методов object.Equals и object.GetHashCode, которые были описаны в главе 6. В следующем примере мы определяем класс Customer с двумя полями и затем записываем компаратор эквивалентности, сопоставляющий имена и фамилии:

```
public class Customer
{
  public string LastName;
  public string FirstName;

  public Customer (string last, string first)
  {
    LastName = last;
    FirstName = first;
  }
}
public class LastFirstEqComparer : EqualityComparer <Customer>
{
  public override bool Equals (Customer x, Customer y)
    => x.LastName == y.LastName && x.FirstName == y.FirstName;

  public override int GetHashCode (Customer obj)
    => (obj.LastName + ";" + obj.FirstName).GetHashCode();
}
```

Чтобы проиллюстрировать его работу, мы создадим два экземпляра класса Customer:

```
Customer c1 = new Customer ("Bloggs", "Joe");
Customer c2 = new Customer ("Bloggs", "Joe");
```

Поскольку метод object.Equals не был переопределен, применяется нормальная семантика эквивалентности ссылочных типов:

```
Console.WriteLine (c1 == c2);              // False
Console.WriteLine (c1.Equals (c2));        // False
```

Та же самая стандартная семантика эквивалентности применяется, когда эти экземпляры используются в классе Dictionary без указания компаратора эквивалентности:

```
var d = new Dictionary<Customer, string>();
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));    // False
```

А теперь укажем специальный компаратор эквивалентности:

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));    // True
```

В приведенном примере необходимо проявлять осторожность, чтобы не изменить значение полей FirstName или LastName экземпляра Customer пока с ним производится работа в словаре, иначе изменится его хеш-код и функционирование словаря будет нарушено.

### EqualityComparer<T>.Default

Свойство EqualityComparer<T>.Default возвращает универсальный компаратор эквивалентности, который может использоваться в качестве альтернативы вызову статического метода object.Equals. Его преимущество заключается в том, что он сначала проверяет, реализует ли тип T интерфейс IEquatable<T>, и если реализует, то вызывает данную реализацию, избегая накладных расходов на упаковку. Это особенно удобно в обобщенных методах:

```
static bool Foo<T> (T x, T y)
{
  bool same = EqualityComparer<T>.Default.Equals (x, y);
  ...
}
```

## IComparer и Comparer

Компараторы используются для переключения на специальную логику упорядочения в отсортированных словарях и коллекциях.

Обратите внимание, что компаратор бесполезен в несортированных словарях, таких как Dictionary и Hashtable – они требуют реализации IEqualityComparer для получения хеш-кодов. Подобным же образом компаратор эквивалентности бесполезен для отсортированных словарей и коллекций.

Ниже приведены определения интерфейса IComparer:

```
public interface IComparer
{
  int Compare(object x, object y);
}
public interface IComparer <in T>
{
  int Compare(T x, T y);
}
```

Как и в случае компараторов эквивалентности, имеется абстрактный класс, предназначенный для создания из него подклассов вместо реализации указанных интерфейсов:

```
public abstract class Comparer<T> : IComparer, IComparer<T>
{
   public static Comparer<T> Default { get; }
   public abstract int Compare (T x, T y);        // Реализуется вами
   int IComparer.Compare (object x, object y);    // Реализован для вас
}
```

В следующем примере показан класс, который описывает желание (wish), и компаратор, сортирующий желания по приоритету:

```
class Wish
{
  public string Name;
  public int Priority;
  public Wish (string name, int priority)
  {
    Name = name;
    Priority = priority;
  }
}
```

```
class PriorityComparer : Comparer <Wish>
{
  public override int Compare (Wish x, Wish y)
  {
    if (object.Equals (x, y)) return 0;  // Отказоустойчивая проверка
    return x.Priority.CompareTo (y.Priority);
  }
}
```

Вызов метода `object.Equals` гарантирует, что путаница с методом `Equals` никогда не возникнет. В данном случае вызов статического метода `object.Equals` лучше вызова `x.Equals`, потому что он будет работать, даже если `x` равно `null`!

Ниже показано, как применять класс `PriorityComparer` для сортировки содержимого `List`:

```
var wishList = new List<Wish>();
wishList.Add (new Wish ("Peace", 2));
wishList.Add (new Wish ("Wealth", 3));
wishList.Add (new Wish ("Love", 2));
wishList.Add (new Wish ("3 more wishes", 1));

wishList.Sort (new PriorityComparer());
foreach (Wish w in wishList) Console.Write (w.Name + " | ");

// ВЫВОД: 3 more wishes | Love | Peace | Wealth |
```

В следующем примере класс `SurnameComparer` позволяет сортировать строки фамилий в порядке, подходящем для телефонной книги:

```
class SurnameComparer : Comparer <string>
{
  string Normalize (string s)
  {
    s = s.Trim().ToUpper();
    if (s.StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }

  public override int Compare (string x, string y)
    => Normalize (x).CompareTo (Normalize (y));
}
```

А вот как использовать класс `SurnameComparer` в отсортированном словаре:

```
var dic = new SortedDictionary<string,string> (new SurnameComparer());
dic.Add ("MacPhail", "second!");
dic.Add ("MacWilliam", "third!");
dic.Add ("McDonald", "first!");

foreach (string s in dic.Values)
  Console.Write (s + " ");          // first! second! third!
```

## StringComparer

`StringComparer` — предопределенный подключаемый класс для поддержки эквивалентности и сравнения строк, который позволяет указывать язык и чувствительность к регистру символов. Он реализует интерфейсы `IEqualityComparer` и `IComparer` (а также их обобщенные версии), так что может применяться с любым типом словаря или отсортированной коллекции:

```
// Класс CultureInfo определен в пространстве имен System.Globalization
public abstract class StringComparer : IComparer, IComparer <string>,
                                       IEqualityComparer,
                                       IEqualityComparer <string>
{
  public abstract int Compare (string x, string y);
  public abstract bool Equals (string x, string y);
  public abstract int GetHashCode (string obj);

  public static StringComparer Create (CultureInfo culture, bool ignoreCase);
  public static StringComparer CurrentCulture { get; }
  public static StringComparer CurrentCultureIgnoreCase { get; }
  public static StringComparer InvariantCulture { get; }
  public static StringComparer InvariantCultureIgnoreCase { get; }
  public static StringComparer Ordinal { get; }
  public static StringComparer OrdinalIgnoreCase { get; }
}
```

Из-за того, что класс `StringComparer` является абстрактным, экземпляры получаются через его статические методы и свойства. Свойство `StringComparer.Ordinal` отражает стандартное поведение для строкового сравнения эквивалентности, а свойство `StringComparer.CurrentCulture` – стандартное поведение для сравнения порядка.

В следующем примере создается ординальный нечувствительный к регистру словарь, в рамках которого `dict["Joe"]` и `dict["JOE"]` означают то же самое:

```
var dict = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

Ниже показано, как отсортировать массив имен с использованием австралийского английского:

```
string[] names = { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo ("en-AU");
Array.Sort<string> (names, StringComparer.Create (ci, false));
```

В финальном примере представлена учитывающая культуру версия класса `SurnameComparer`, написанного в предыдущем разделе (со сравнением имен, подходящим для телефонной книги):

```
class SurnameComparer : Comparer <string>
{
  StringComparer strCmp;

  public SurnameComparer (CultureInfo ci)
  {
    // Создать строковый компаратор, чувствительный к регистру и культуре
    strCmp = StringComparer.Create (ci, false);
  }
  string Normalize (string s)
  {
    s = s.Trim();
    if (s.ToUpper().StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }
  public override int Compare (string x, string y)
  {
    // Напрямую вызвать Compare на учитывающем культуру StringComparer
    return strCmp.Compare (Normalize (x), Normalize (y));
  }
}
```

## IStructuralEquatable и IStructuralComparable

Как утверждалось в предыдущей главе, по умолчанию структуры реализуют *структурное сравнение*: две структуры эквивалентны, если эквивалентны все их поля. Однако иногда структурная эквивалентность и сравнение порядка удобны в виде подключаемых вариантов также для других типов, таких как массивы и кортежи. Для этих целей в .NET Framework 4.0 были введены два новых интерфейса:

```
public interface IStructuralEquatable
{
  bool Equals (object other, IEqualityComparer comparer);
  int GetHashCode (IEqualityComparer comparer);
}
public interface IStructuralComparable
{
  int CompareTo (object other, IComparer comparer);
}
```

Передаваемая реализация `IEqualityComparer/IComparer` применяется к каждому индивидуальному элементу в составном объекте. Мы можем продемонстрировать это с использованием массивов и кортежей, которые реализуют указанные интерфейсы. В следующем примере мы сравниваем два массива на предмет эквивалентности сначала с применением стандартного метода `Equals`, а затем его версии из интерфейса `IStructuralEquatable`:

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };
IStructuralEquatable se1 = a1;
Console.Write (a1.Equals (a2));                                    // False
Console.Write (se1.Equals (a2, EqualityComparer<int>.Default));    // True
```

Вот еще один пример:

```
string[] a1 = "the quick brown fox".Split();
string[] a2 = "THE QUICK BROWN FOX".Split();
IStructuralEquatable se1 = a1;
bool isTrue = se1.Equals (a2, StringComparer.InvariantCultureIgnoreCase);
```

# 8

# Запросы LINQ

Язык интегрированных запросов (Language Integrated Query – LINQ) представляет собой набор языковых и платформенных средств для написания структурированных, безопасных в отношении типов запросов к локальным коллекциям объектов и удаленным источникам данных. LINQ появился в C# 3.0 и .NET Framework 3.5.

Язык LINQ позволяет создавать запросы к любой коллекции, которая реализует интерфейс `IEnumerable<T>`, будь то массив, список или DOM-модель XML, а также к удаленным источникам данных, таким как таблицы в базе данных SQL Server. Язык LINQ обладает преимуществами проверки типов на этапе компиляции и формирования динамических запросов.

В настоящей главе объясняется архитектура LINQ и фундаментальные основы написания запросов. Все основные типы определены в пространствах имен `System.Linq` и `System.Linq.Expressions`.

Примеры, рассматриваемые в текущей и двух последующих главах, доступны вместе с интерактивным инструментом запросов под названием LINQPad. Загрузить LINQPad можно на веб-сайте `www.linqpad.net`.

## Начало работы

Базовыми единицами данных в LINQ являются *последовательности* и *элементы*. Последовательность – это любой объект, который реализует интерфейс `IEnumerable<T>`, а элемент – это каждая единица данных внутри последовательности. В следующем примере `names` является последовательностью, а `"Tom"`, `"Dick"` и `"Harry"` – элементами:

```
string[] names = { "Tom", "Dick", "Harry" };
```

Мы называем ее *локальной последовательностью*, потому что она представляет локальную коллекцию объектов в памяти.

*Операция запроса* – это метод, который трансформирует последовательность. Типичная операция запроса принимает *входную последовательность* и выдает трансформированную *выходную последовательность*. В классе `Enumerable` из пространства имен `System.Linq` имеется около 40 операций запросов – все они реализованы в виде статических методов. Их называют *стандартными операциями запросов*.

Запросы, оперирующие на локальных последовательностях, называются локальными запросами или запросами *LINQ to Objects*. Язык LINQ также поддерживает последовательности, которые могут динамически наполняться из удаленного источника данных, такого как база данных SQL Server. Последовательности подобного рода дополнительно реализуют интерфейс `IQueryable<T>` и поддерживаются через соответствующий набор стандартных операций запросов в классе `Queryable`. Мы обсудим данную тему более подробно в разделе "Интерпретируемые запросы" далее в главе.

Запрос представляет собой выражение, которое при перечислении трансформирует последовательности с помощью операций запросов. Простейший запрос состоит из одной входной последовательности и одной операции. Например, мы можем применить операцию `Where` к простому массиву для извлечения элементов с длиной, по меньшей мере, четыре символа:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.WriteLine (n);
```

```
Dick
Harry
```

Поскольку стандартные операции запросов реализованы в виде расширяющих методов, мы можем вызывать `Where` прямо на `names` — как если бы он был методом экземпляра:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

Чтобы такой код скомпилировался, потребуется импортировать пространство имен `System.Linq`. Ниже приведен завершенный пример:

```
using System;
usign System.Collections.Generic;
using System.Linq;
class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry" };
    IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
    foreach (string name in filteredNames) Console.WriteLine (name);
  }
}
```

```
Dick
Harry
```

Мы могли бы дополнительно сократить код, неявно типизируя `filteredNames`:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

Однако в результате затрудняется зрительное восприятие запроса, особенно за пределами IDE-среды, где нет никаких всплывающих подсказок, которые помогли бы понять запрос.

В настоящей главе мы будем избегать неявной типизации результатов запросов кроме тех случаев, когда она обязательна (см. раздел "Стратегии проекции" далее в главе) или когда тип запроса не связан с примером.

Большинство операций запросов принимают в качестве аргумента лямбда-выражение. Лямбда-выражение помогает направлять и формировать запрос. В приведенном выше примере лямбда-выражение выглядит следующим образом:

```
n => n.Length >= 4
```

Входной аргумент соответствует входному элементу. В рассматриваемой ситуации входной аргумент `n` представляет каждое имя в массиве и относится к типу `string`. Операция `Where` требует, чтобы лямбда-выражение возвращало значение `bool`, которое в случае равенства `true` указывает на то, что элемент должен быть включен в выходную последовательность. Ниже приведена его сигнатура:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

Следующий запрос извлекает все имена, которые содержат букву "a":

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains ("a"));

foreach (string name in filteredNames)
  Console.WriteLine (name);  // Harry
```

До сих пор мы строили запросы, используя расширяющие методы и лямбда-выражения. Как вскоре будет показано, такая стратегия хорошо компонуема в том смысле, что позволяет формировать цепочки операций запросов. В книге мы будем называть это *текучим синтаксисом*[1]. Для написания запросов язык C# также предлагает другой синтаксис, который называется синтаксисом *выражений запросов*. Вот как выглядит предыдущий запрос, записанный как выражение запроса:

```
IEnumerable<string> filteredNames = from n in names
                                    where n.Contains ("a")
                                    select n;
```

Текучий синтаксис и синтаксис выражений запросов дополняют друг друга. В следующих двух разделах мы исследуем каждый из них более детально.

# Текучий синтаксис

Текучий синтаксис является наиболее гибким и фундаментальным. В этом разделе мы покажем, как выстраивать цепочки операций для формирования более сложных запросов, а также объясним важность расширяющих методов в данном процессе. Мы также расскажем, как формулировать лямбда-выражения для операции запроса и представим несколько новых операций запросов.

## Выстраивание в цепочки операций запросов

В предыдущем разделе были показаны два простых запроса, включающие одну операцию. Для построения более сложных запросов к выражению добавляются дополнительные операции запросов, формируя в итоге цепочку. В качестве примера следующий запрос извлекает все строки, содержащие букву "a", сортирует их по длине и затем преобразует результаты в верхний регистр:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

[1] Термин "текучий" (fluent) основан на работе Эрика Эванса и Мартина Фаулера, посвященной текучим интерфейсам.

```
class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

    IEnumerable<string> query = names
      .Where   (n => n.Contains ("a"))
      .OrderBy (n => n.Length)
      .Select  (n => n.ToUpper());

    foreach (string name in query) Console.WriteLine (name);
  }
}
```

```
JAY
MARY
HARRY
```

В приведенном примере переменная n имеет закрытую область видимости в каждом лямбда-выражении. Переменную n можно многократно использовать по той же причине, по которой подобное возможно для переменной c в следующем методе:

```
void Test()
{
  foreach (char c in "string1") Console.Write (c);
  foreach (char c in "string2") Console.Write (c);
  foreach (char c in "string3") Console.Write (c);
}
```

`Where`, `OrderBy` и `Select` — стандартные операции запросов, которые распознаются как вызовы расширяющих методов класса `Enumerable` (если импортировано пространство имен `System.Linq`).

Мы уже представляли операцию `Where`, которая выдает отфильтрованную версию входной последовательности. Операция `OrderBy` выдает отсортированную версию своей входной последовательности, а метод `Select` — последовательность, в которой каждый входной элемент трансформирован, или *спроецирован*, с помощью заданного лямбда-выражения (`n.ToUpper` в этом случае). Данные протекают слева направо через цепочку операций, так что они сначала фильтруются, затем сортируются и, наконец, проецируются.

Операция запроса никогда не изменяет входную последовательность; взамен она возвращает новую последовательность. Подход согласуется с парадигмой *функционального программирования*, которая была побудительной причиной создания языка LINQ.

Ниже приведены сигнатуры задействованных ранее расширяющих методов (с несколько упрощенной сигнатурой `OrderBy`):

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)

public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Когда операции запросов выстраиваются в цепочку, как в рассмотренном примере, выходная последовательность одной операции является входной последовательностью следующей операции. Полный запрос напоминает производственную линию с конвейерными лентами (рис. 8.1).

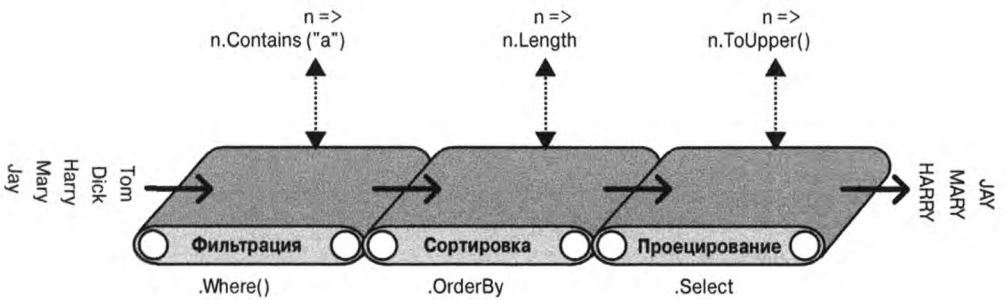



***Рис. 8.1.** Цепочка операций запросов*

Точно такой же запрос можно строить *постепенно*, как показано ниже:

```
// Для компиляции этого кода потребуется импортировать пространство имен
System.Linq:

IEnumerable<string> filtered  = names   .Where    (n => n.Contains ("a"));
IEnumerable<string> sorted    = filtered.OrderBy (n => n.Length);
IEnumerable<string> finalQuery = sorted  .Select   (n => n.ToUpper());
```

Запрос `finalQuery` композиционно идентичен ранее сконструированному запросу `query`. Кроме того, каждый промежуточный шаг также состоит из допустимого запроса, который можно выполнить:

```
foreach (string name in filtered)
  Console.Write (name + "|");       // Harry|Mary|Jay|

Console.WriteLine();
foreach (string name in sorted)
  Console.Write (name + "|");       // Jay|Mary|Harry|

Console.WriteLine();
foreach (string name in finalQuery)
  Console.Write (name + "|");       // JAY|MARY|HARRY|
```

## Почему расширяющие методы важны

Вместо применения синтаксиса расширяющих методов для работы с операциями запросов можно использовать привычный синтаксис статических методов. Например:

```
IEnumerable<string> filtered = Enumerable.Where (names,
                                                 n => n.Contains ("a"));
IEnumerable<string> sorted = Enumerable.OrderBy (filtered, n => n.Length);
IEnumerable<string> finalQuery = Enumerable.Select (sorted,
                                                    n => n.ToUpper());
```

В действительности именно так компилятор транслирует вызовы расширяющих методов. Однако избегание расширяющих методов не обходится даром, когда жела-

тельно записать запрос в одном операторе, как делалось ранее. Давайте вернемся к запросу в виде одного оператора — сначала с синтаксисом расширяющих методов:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                  .OrderBy (n => n.Length)
                                  .Select  (n => n.ToUpper());
```

Его естественная линейная форма отражает протекание данных слева направо, а также удерживает лямбда-выражения рядом с их операциями запросов (*инфиксная* система обозначений). Без расширяющих методов запрос теряет свою *текучесть*:

```
IEnumerable<string> query =
  Enumerable.Select (
    Enumerable.OrderBy (
      Enumerable.Where (
        names, n => n.Contains ("a")
      ), n => n.Length
    ), n => n.ToUpper()
  );
```

## Составление лямбда-выражений

В предыдущем примере мы передаем операции `Where` следующее лямбда-выражение:

```
n => n.Contains ("a")  // Входной тип - string, возвращаемый тип - bool
```

Лямбда-выражение, которое принимает значение и возвращает результат типа `bool`, называется *предикатом*.

Назначение лямбда-выражения зависит от конкретной операции запроса. В операции `Where` оно указывает, должен ли элемент помещаться в выходную последовательность. В случае операции `OrderBy` лямбда-выражение отображает каждый элемент во входной последовательности на его ключ сортировки. В операции `Select` лямбда-выражение определяет, каким образом каждый элемент во входной последовательности трансформируется перед попаданием в выходную последовательность.

Лямбда-выражение в операции запроса всегда работает на индивидуальных элементах во входной последовательности, но не на последовательности как едином целом.

Операция запроса вычисляет лямбда-выражение по требованию — обычно один раз на элемент во входной последовательности. Лямбда-выражения позволяют помещать внутрь операций запросов собственную логику. Это делает операции запросов универсальными — и одновременно простыми по своей сути. Ниже приведена полная реализация `Enumerable.Where` кроме обработки исключений:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

### Лямбда-выражения и сигнатуры Func

Стандартные операции запросов задействуют обобщенные делегаты Func. Семейство универсальных обобщенных делегатов под названием Func определено в пространстве имен System со следующим замыслом.

> *Аргументы типа в Func появляются в том же самом порядке, что и в лямбда-выражениях.*

Таким образом, Func<TSource,bool> соответствует лямбда-выражению TSource=>bool, которое принимает аргумент TSource и возвращает значение bool.

Аналогично Func<TSource,TResult> соответствует лямбда-выражению TSource=>TResult.

Делегаты Func перечислены в разделе "Делегаты Func и Action" главы 4.

### Лямбда-выражения и типизация элементов

Стандартные операции запросов применяют описанные ниже имена обобщенных типов.

| Имя обобщенного типа | Что означает |
|---|---|
| TSource | Тип элемента для входной последовательности |
| TResult | Тип элемента для выходной последовательности, если он отличается от TSource |
| TKey | Тип элемента для ключа, используемого при сортировке, группировании или соединении |

Тип TSource определяется входной последовательностью. Типы TResult и TKey обычно *выводятся из лямбда-выражения.*

Например, взгляните на сигнатуру операции запроса Select:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Делегат Func<TSource,TResult> соответствует лямбда-выражению TSource=> TResult, которое отображает *входной элемент* на *выходной элемент.* Типы TSource и TResult могут быть разными, так что лямбда-выражение способно изменять тип каждого элемента. Более того, лямбда-выражение *определяет тип выходной последовательности.* В следующем запросе с помощью операции Select выполняется трансформация элементов строкового типа в элементы целочисленного типа:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<int> query = names.Select (n => n.Length);
foreach (int length in query)
  Console.Write (length + "|");  // 3|4|5|4|3|
```

Компилятор может *выводить* тип TResult из возвращаемого значения лямбда-выражения. В данном случае n.Length возвращает значение int, поэтому для TResult выводится тип int.

Операция запроса Where проще и не требует выведения типа для выходной последовательности, т.к. входной и выходной элементы относятся к тому же самому типу. Это имеет смысл, потому что данная операция просто фильтрует элементы; она не *трансформирует* их:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

Наконец, рассмотрим сигнатуру операции OrderBy:

```
// Несколько упрощена:
public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)
```

Делегат Func<TSource,TKey> отображает входной элемент на *ключ сортировки*. Тип TKey выводится из лямбда-выражения и является отдельным от типов входного и выходного элементов. Например, можно реализовать сортировку списка имен по длине (ключ int) или в алфавитном порядке (ключ string):

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength      = names.OrderBy (n => n.Length);      // ключ int
sortedAlphabetically = names.OrderBy (n => n);            // ключ string
```

Операции запросов в классе Enumerable можно вызывать с помощью традиционных делегатов, ссылающихся на методы, а не на лямбда-выражения. Такой подход эффективен в плане упрощения некоторых видов локальных запросов – особенно LINQ to XML – и демонстрируется в главе 10. Однако он не работает с последовательностями, основанными на интерфейсе IQueryable<T> (например, при запрашивании базы данных), т.к. операции в классе Queryable требуют лямбда-выражений для выпуска деревьев выражений. Мы обсудим это позже в разделе "Интерпретируемые запросы".

## Естественный порядок

Исходный порядок элементов внутри входной последовательности в языке LINQ важен. На такое поведение полагаются некоторые операции запросов, в частности, Take, Skip и Reverse. Операция Take выдает первые x элементов, отбрасывая остальные:

```
int[] numbers  = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take (3);  // { 10, 9, 8 }
```

Операция Skip игнорирует первые x элементов и выдает остальные:

```
IEnumerable<int> lastTwo    = numbers.Skip (3);  // { 7, 6 }
```

Операция Reverse изменяет порядок следования элементов на противоположный:

```
IEnumerable<int> reversed   = numbers.Reverse();  // { 6, 7, 8, 9, 10 }
```

В локальных запросах (LINQ to Objects) операции наподобие Where и Select предохраняют исходный порядок во входной последовательности (как поступают все остальные операции запросов за исключением тех, которые специально изменяют порядок).

## Другие операции

Не все операции запросов возвращают последовательность. Операции над *элементами* извлекают один элемент из входной последовательности; примерами таких операций служат First, Last, Single и ElementAt:

```
int[] numbers    = { 10, 9, 8, 7, 6 };
int firstNumber  = numbers.First();                          // 10
int lastNumber   = numbers.Last();                           // 6
int secondNumber = numbers.ElementAt(1);                     // 9
int secondLowest = numbers.OrderBy(n=>n).Skip(1).First(); // 7
```

Операции *агрегирования* возвращают скалярное значение, обычно числового типа:

```
int count = numbers.Count();  // 5;
int min = numbers.Min();      // 6;
```

*Квантификаторы* возвращают значение `bool`:

```
bool hasTheNumberNine = numbers.Contains (9);         // true
bool hasMoreThanZeroElements = numbers.Any();         // true
bool hasAnOddElement = numbers.Any (n => n % 2 != 0); // true
```

Поскольку эти операции возвращают одиночный элемент, вызов дополнительных операций запросов на их результате обычно не производится, если только сам элемент не является коллекцией.

Некоторые операции запросов принимают две входных последовательности. Примерами могут служить операция `Concat`, которая добавляет одну последовательность к другой, и операция `Union`, делающая то же самое, но с удалением дубликатов:

```
int[] seq1 = { 1, 2, 3 };
int[] seq2 = { 3, 4, 5 };
IEnumerable<int> concat = seq1.Concat (seq2);    // { 1, 2, 3, 3, 4, 5 }
IEnumerable<int> union  = seq1.Union (seq2);     // { 1, 2, 3, 4, 5 }
```

К данной категории относятся также и операции соединения. Все операции запросов подробно рассматриваются в главе 9.

# Выражения запросов

Язык C# предоставляет синтаксическое сокращение для написания запросов LINQ, которое называется *выражениями запросов*. Вопреки распространенному мнению выражение запроса не является средством встраивания в C# возможностей языка SQL. В действительности на проектное решение для выражений запросов повлияли главным образом *генераторы списков* (list comprehension) из таких языков функционального программирования, как LISP и Haskell, хотя косметическое влияние оказал и язык SQL.

В настоящей книге мы ссылаемся на синтаксис выражений запросов просто как на "синтаксис запросов".

В предыдущем разделе с использованием текучего синтаксиса мы написали запрос для извлечения строк, содержащих букву "a", их сортировки и преобразования в верхний регистр. Ниже показано, как сделать то же самое с помощью синтаксиса запросов:

```
using System;
using System.Collections.Generic;
using System.Linq;

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

    IEnumerable<string> query =
      from   n in names
      where  n.Contains ("a") // Фильтровать элементы
      orderby n.Length        // Сортировать элементы
      select  n.ToUpper();    // Транслировать (проецировать) каждый элемент
```

```
    foreach (string name in query) Console.WriteLine (name);
  }
}
JAY
MARY
HARRY
```

Выражения запросов всегда начинаются с конструкции `from` и заканчиваются либо конструкцией `select`, либо конструкцией `group`. Конструкция `from` объявляет *переменную диапазона* (в данном случае `n`), которую можно воспринимать как переменную, предназначенную для обхода входной последовательности – почти как в цикле `foreach`. На рис. 8.2 представлен полный синтаксис в виде синтаксической диаграммы.

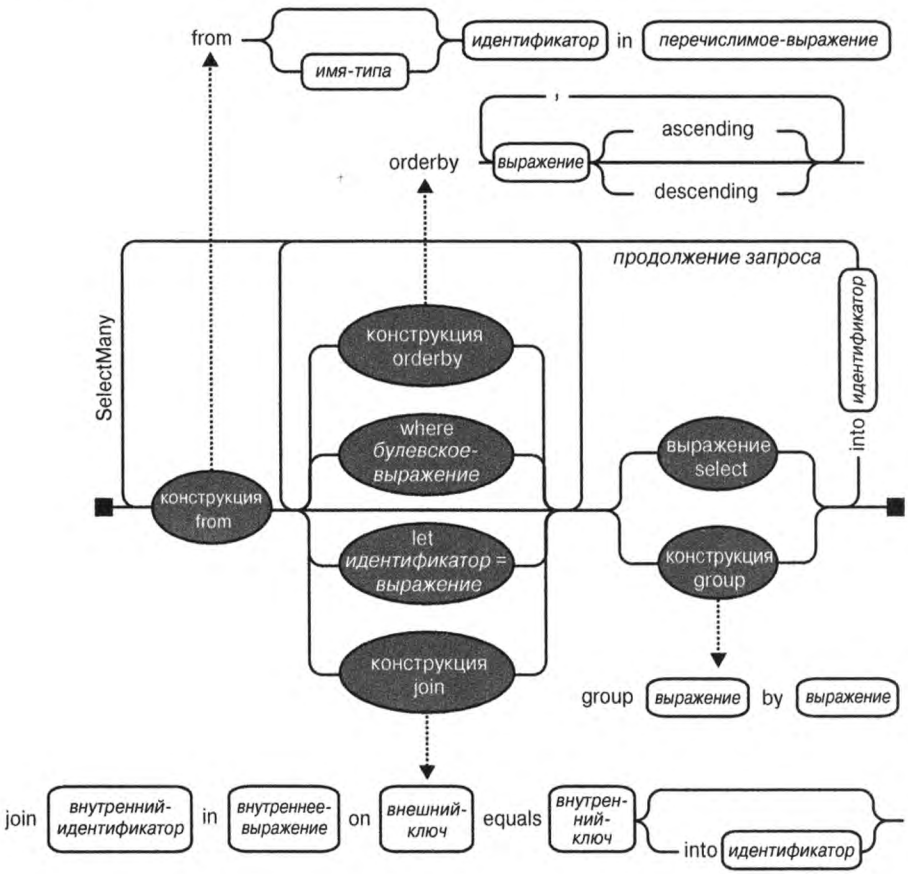



***Рис. 8.2.** Синтаксис запросов*

Начинайте читать диаграмму слева и продолжайте двигаться по пути подобно поезду. Например, после обязательной конструкции `from` можно дополнительно включить конструкцию `orderby`, `where`, `let` или `join`. После этого можно либо продолжить конструкцией `select` или `group`, либо вернуться и включить еще одну конструкцию `from`, `orderby`, `where`, `let` или `join`.

Компилятор обрабатывает выражения запросов, транслируя их в текучий синтаксис. Трансляция делается в довольно-таки механической манере – очень похоже на то, как операторы `foreach` транслируются в вызовы методов `GetEnumerator` и `MoveNext`. Это значит, что любое выражение запроса, написанное с применением синтаксиса запросов, можно также представить посредством текучего синтаксиса. Компилятор (первоначально) транслирует показанный выше пример запроса в следующий код:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                  .OrderBy (n => n.Length)
                                  .Select  (n => n.ToUpper());
```

Затем операции `Where`, `OrderBy` и `Select` преобразуются с использованием тех же правил, которые применялись бы к запросу, написанному с помощью текучего синтаксиса. В данном случае операции привязываются к расширяющим методам в классе `Enumerable`, т.к. пространство имен `System.Linq` импортировано и тип `names` реализует интерфейс `IEnumerable<string>`. Однако при трансляции выражений запросов компилятор не оказывает специальное содействие классу `Enumerable`. Можете считать, что компилятор механически вводит слова `Where`, `OrderBy` и `Select` внутрь оператора, после чего компилирует его, как если бы вы набирали такие имена методов самостоятельно. Это обеспечивает гибкость в том, как они распознаются. Например, операции в запросах к базе данных, которые мы будем строить в последующих разделах, взамен привязываются к расширяющим методам из класса `Queryable`.

Если удалить из программы директиву `using System.Linq`, тогда запросы не скомпилируются, поскольку методы `Where`, `OrderBy` и `Select` не к чему привязывать. Выражения запросов *не могут быть скомпилированы* до тех пор, пока не будет импортировано `System.Linq` или другое пространство имен с реализацией этих методов запросов.

## Переменные диапазона

Идентификатор, непосредственно следующий за словом `from` в синтаксисе, называется *переменной диапазона*. Переменная диапазона ссылается на текущий элемент в последовательности, в отношении которой должна выполняться операция.

В наших примерах переменная диапазона `n` присутствует в каждой конструкции запроса. Однако в каждой конструкции эта переменная на самом деле выполняет перечисление *разных* последовательностей:

```
from    n in names        // n - переменная диапазона
where   n.Contains ("a")  // n берется прямо из массива
orderby n.Length          // n впоследствии фильтруется
select  n.ToUpper()       // n впоследствии сортируется
```

Все станет ясным, если взглянуть на механическую трансляцию в текучий синтаксис, предпринимаемую компилятором:

```
names.Where   (n => n.Contains ("a"))   // n с локальной областью видимости
     .OrderBy (n => n.Length)           // n с локальной областью видимости
     .Select  (n => n.ToUpper())        // n с локальной областью видимости
```

Как видите, каждый экземпляр переменной `n` имеет закрытую область видимости, которая ограничена собственным лямбда-выражением.

Выражения запросов также позволяют вводить новые переменные диапазонов с помощью следующих конструкций:

- `let`
- `into`
- дополнительная конструкция `from`
- `join`

Мы рассмотрим данную тему позже в разделе "Стратегии композиции" настоящей главы и также в разделах "Выполнение проекции" и "Выполнение соединения" главы 9.

## Сравнение синтаксиса запросов и синтаксиса SQL

Выражения запросов внешне похожи на код SQL, хотя они существенно отличаются. Запрос LINQ сводится к выражению C#, а потому следует стандартным правилам языка C#. Например, в LINQ нельзя использовать переменную до ее объявления. Язык SQL разрешает ссылаться в операторе `SELECT` на псевдоним таблицы до его определения в конструкции `FROM`.

Подзапрос в LINQ является просто еще одним выражением C#, так что никакого специального синтаксиса он не требует. Подзапросы в SQL подчиняются специальным правилам.

В LINQ данные логически протекают слева направо через запрос. Что касается потока данных в SQL, то порядок структурирован не настолько хорошо.

Запрос LINQ состоит из *конвейера* операций, принимающих и выпускающих последовательности, в которых порядок следования элементов может иметь значение. Запрос SQL образован из *сети* конструкций, которые работают по большей части с *неупорядоченными наборами*.

## Сравнение синтаксиса запросов и текучего синтаксиса

И синтаксис выражений запросов, и текучий синтаксис обладают своими преимуществами.

Синтаксис запросов проще для запросов, которые содержат в себе любой из перечисленных ниже аспектов:

- конструкцию `let` для введения новой переменной наряду с переменной диапазона;
- операцию `SelectMany`, `Join` или `GroupJoin`, за которой следует ссылка на внешнюю переменную диапазона.

(Мы опишем конструкцию `let` в разделе "Стратегии композиции" далее в главе, а операции `SelectMany`, `Join` и `GroupJoin` – в главе 9.)

Посредине находятся запросы, которые просто применяют операции `Where`, `OrderBy` и `Select`. С ними одинаково хорошо работает любой синтаксис; выбор здесь определяется в основном персональными предпочтениями.

Для запросов, состоящих из одной операции, текучий синтаксис короче и характеризуется меньшим беспорядком.

Наконец, есть много операций, для которых ключевые слова в синтаксисте запросов не предусмотрены. Такие операции требуют использования текучего синтаксиса – по крайней мере, частично. К ним относится любая операция, выходящая за рамки перечисленных ниже:

```
Where, Select, SelectMany
OrderBy, ThenBy, OrderByDescending, ThenByDescending
GroupBy, Join, GroupJoin
```

## Запросы со смешанным синтаксисом

Когда операция запроса не поддерживается в синтаксисе запросов, синтаксис запросов и текучий синтаксис можно смешивать. Единственное ограничение заключается в том, что каждый компонент синтаксиса запросов должен быть завершен (т.е. начинаться с конструкции `from` и заканчиваться конструкцией `select` или `group`).

Предположим, что есть такое объявление массива:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

В следующем примере подсчитывается количество имен, содержащих букву "a":

```
int matches = (from n in names where n.Contains ("a") select n).Count(); // 3
```

Показанный ниже запрос получает имя в алфавитном порядке:

```
string first = (from n in names orderby n select n).First();  // Dick
```

Подход со смешанным синтаксисом иногда полезен в более сложных запросах. Однако в представленных выше простых примерах мы могли бы безо всяких проблем придерживаться текучего синтаксиса:

```
int matches = names.Where (n => n.Contains ("a")).Count();      // 3
string first = names.OrderBy (n => n).First();                  // Dick
```

Временами запросы со смешанным синтаксисом обеспечивают почти максимальную выгоду в плане функциональности и простоты. Важно не отдавать одностороннее предпочтение синтаксису запросов или текучему синтаксису, иначе вы не сможете записывать запросы со смешанным синтаксисом без ощущения того, что допускаете ошибку!

В оставшейся части главы мы будем демонстрировать ключевые концепции с применением обоих видов синтаксиса, когда это уместно.

# Отложенное выполнение

Важная особенность большинства операций запросов связана с тем, что они выполняются не тогда, когда создаются, а когда происходит *перечисление* (другими словами, при вызове метода `MoveNext` на перечислителе). Рассмотрим следующий запрос:

```
var numbers = new List<int>();
numbers.Add (1);

IEnumerable<int> query = numbers.Select (n => n * 10);  // Построить запрос

numbers.Add (2);              // Вставить дополнительный элемент

foreach (int n in query)
  Console.Write (n + "|");    // 10|20|
```

Дополнительное число, вставленное в список *после* конструирования запроса, включается в результат, поскольку любая фильтрация или сортировка не происходит вплоть до выполнения оператора `foreach`. Это называется *отложенным* или *ленивым* выполнением и представляет собой то же самое действие, которое происходит с делегатами:

```
Action a = () => Console.WriteLine ("Foo");
// Пока на консоль ничего не выводится. А теперь запустим запрос:
a();  // Отложенное выполнение!
```

Отложенное выполнение поддерживают все стандартные операции запросов со следующими исключениями:

- операции, которые возвращают одиночный элемент или скалярное значение, такие как `First` или `Count`;
- перечисленные ниже *операции преобразования*:

  `ToArray, ToList, ToDictionary, ToLookup`

Указанные операции вызывают немедленное выполнение запроса, т.к. их результирующие типы не имеют механизма для обеспечения отложенного выполнения. Скажем, метод `Count` возвращает простое целочисленное значение, для которого последующее перечисление невозможно. Показанный ниже запрос выполняется немедленно:

```
int matches = numbers.Where (n => n < 2).Count();  // 1
```

Отложенное выполнение важно из-за того, что оно отвязывает *конструирование* запроса от его *выполнения*. Это позволяет строить запрос в течение нескольких шагов, а также делает возможными запросы к базе данных.

Подзапросы предоставляют еще один уровень косвенности. Все, что находится в подзапросе, подпадает под отложенное выполнение — включая методы агрегирования и преобразования. Мы рассмотрим их в разделе "Подзапросы" далее в главе.

## Повторная оценка

С отложенным выполнением связано еще одно последствие — запрос с отложенным выполнением при повторном перечислении оценивается заново:

```
var numbers = new List<int>() { 1, 2 };
IEnumerable<int> query = numbers.Select (n => n * 10);
foreach (int n in query) Console.Write (n + "|");     // 10|20|
numbers.Clear();
foreach (int n in query) Console.Write (n + "|");     // Ничего не выводится
```

Есть пара причин, по которым повторная оценка иногда неблагоприятна:

- временами требуется "заморозить" или кешировать результаты в определенный момент времени;
- некоторые запросы сопровождаются большим объемом вычислений (или полагаются на обращение к удаленной базе данных), поэтому повторять их без настоятельной необходимости нежелательно.

Повторной оценки можно избежать за счет вызова операции преобразования, такой как `ToArray` или `ToList`. Операция `ToArray` копирует выходные данные запроса в массив, а `ToList` — в обобщенный список `List<T>`:

```
var numbers = new List<int>() { 1, 2 };

List<int> timesTen = numbers
  .Select (n => n * 10)
  .ToList();  // Выполняется немедленное преобразование в List<int>
numbers.Clear();
Console.WriteLine (timesTen.Count);  // По-прежнему 2
```

## Захваченные переменные

Если лямбда-выражения запроса *захватывают* внешние переменные, то запрос будет принимать значения таких переменных в момент, когда он *запускается*:

```
int[] numbers = { 1, 2 };

int factor = 10;
IEnumerable<int> query = numbers.Select (n => n * factor);
factor = 20;
foreach (int n in query) Console.Write (n + "|");  // 20|40|
```

В итоге может возникнуть проблема при построении запроса внутри цикла `for`. Например, предположим, что необходимо удалить все гласные из строки. Следующий код, несмотря на свою неэффективность, дает корректный результат:

```
IEnumerable<char> query = "Not what you might expect";

query = query.Where (c => c != 'a');
query = query.Where (c => c != 'e');
query = query.Where (c => c != 'i');
query = query.Where (c => c != 'o');
query = query.Where (c => c != 'u');

foreach (char c in query) Console.Write (c);  // Nt wht y mght xpct
```

А теперь посмотрим, что произойдет, если мы переделаем код с использованием цикла `for`:

```
IEnumerable<char> query = "Not what you might expect";
string vowels = "aeiou";

for (int i = 0; i < vowels.Length; i++)
  query = query.Where (c => c != vowels[i]);

foreach (char c in query) Console.Write (c);
```

При перечислении запроса генерируется исключение `IndexOutOfRangeException`, потому что, как было указано в разделе "Захватывание внешних переменных" главы 4, компилятор назначает переменной итерации в цикле `for` такую же область видимости, как если бы она была объявлена *вне* цикла. Следовательно, каждое замыкание захватывает *ту же самую* переменную (`i`), значение которой равно 5, когда начинается действительное перечисление запроса. Чтобы решить проблему, переменную цикла потребуется присвоить другой переменной, объявленной *внутри* блока операторов:

```
for (int i = 0; i < vowels.Length; i++)
{
  char vowel = vowels[i];
  query = query.Where (c => c != vowel);
}
```

В таком случае на каждой итерации цикла будет захватываться свежая локальная переменная.

Начиная с версии C# 5.0, стал доступным еще один способ решения описанной проблемы — замена цикла `for` циклом `foreach`:

```
foreach (char vowel in vowels)
  query = query.Where (c => c != vowel);
```

Приведенный выше код работает в C# 5.0, но не в предшествующих версиях языка, по причинам, которые объяснялись в главе 4.

## Как работает отложенное выполнение

Операции запросов обеспечивают отложенное выполнение за счет возвращения *декораторных* последовательностей.

В отличие от традиционного класса коллекции, такого как массив или связный список, декораторная последовательность (в общем случае) не имеет собственной поддерживающей структуры для хранения элементов. Взамен она является оболочкой для другой последовательности, предоставляемой во время выполнения, и имеет с ней постоянную зависимость. Всякий раз, когда запрашиваются данные из декоратора, он в свою очередь должен запрашивать данные из внутренней входной последовательности.

Трансформация операции запроса образует "декорацию". Если выходная последовательность не подвергается трансформациям, то результатом будет простой *прокси*, а не декоратор.

Вызов операции `Where` просто конструирует декораторную последовательность-оболочку, которая хранит ссылку на входную последовательность, лямбда-выражение и любые другие указанные аргументы. Входная последовательность перечисляется только при перечислении декоратора.

На рис. 8.3 проиллюстрирована композиция следующего запроса:

```
IEnumerable<int> lessThanTen = new int[] { 5, 12, 3 }.Where (n => n < 10);
```

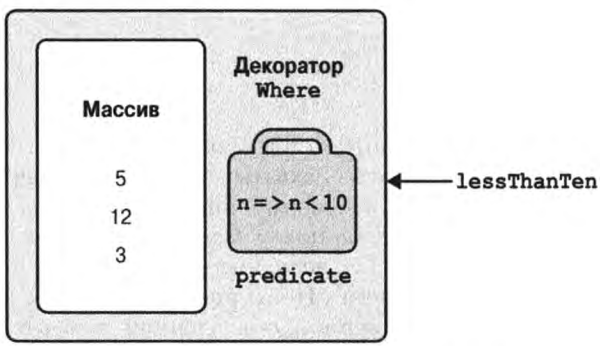



***Рис. 8.3.** Декораторная последовательность*

При перечислении `lessThanTen` в действительности происходит запрос массива через декоратор `Where`.

Хорошая новость заключается в том, что даже если требуется создать собственную операцию запроса, то реализация декораторной последовательности осуществляется просто с помощью итератора C#. Ниже показано, как можно написать собственный метод `Select` (проверка аргументов опущена):

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

Данный метод является итератором благодаря наличию оператора `yield return`. С точки зрения функциональности он представляет собой сокращение для следующего кода:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  return new SelectSequence (source, selector);
}
```

где *SelectSequence* – это (сгенерированный компилятором) класс, перечислитель которого инкапсулирует логику из метода итератора.

Таким образом, при вызове операции вроде `Select` или `Where` всего лишь создается экземпляр перечислимого класса, который декорирует входную последовательность.

## Построение цепочки декораторов

Объединение операций запросов в цепочку приводит к созданию иерархических представлений декораторов. Рассмотрим следующий запрос:

```
IEnumerable<int> query = new int[] { 5, 12, 3 }.Where   (n => n < 10)
                                                .OrderBy (n => n)
                                                .Select  (n => n * 10);
```

Каждая операция запроса создает новый экземпляр декоратора, который является оболочкой для предыдущей последовательности (подобно матрешке). Объектная модель этого запроса показана на рис. 8.4. Обратите внимание, что объектная модель полностью конструируется до выполнения любого перечисления.

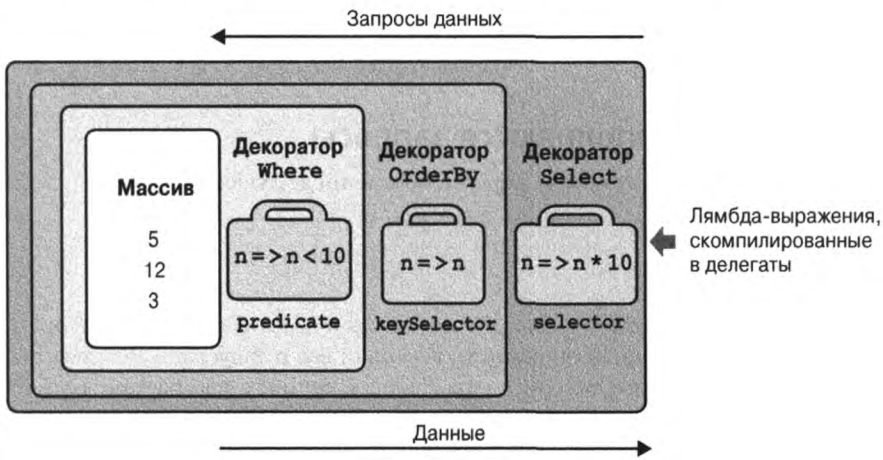



***Рис. 8.4.** Иерархия декораторных последовательностей*

При перечислении `query` производятся запросы к исходному массиву, трансформированному посредством иерархии или цепочки декораторов.

Добавление `ToList` в конец этого запроса приведет к немедленному выполнению предшествующих операций, что свернет всю объектную модель в единственный список.

На рис. 8.5 показана та же композиция объектов в виде UML-диаграммы. Декоратор `Select` ссылается на декоратор `OrderBy`, который в свою очередь ссылается на декоратор `Where`, а тот – на массив. Особенность отложенного выполнения заключается в том, что при постепенном формировании запроса строится идентичная объектная модель:

```
IEnumerable<int>
  source   = new int[] { 5, 12, 3 },
  filtered = source   .Where   (n => n < 10),
  sorted   = filtered .OrderBy (n => n),
  query    = sorted   .Select  (n => n * 10);
```

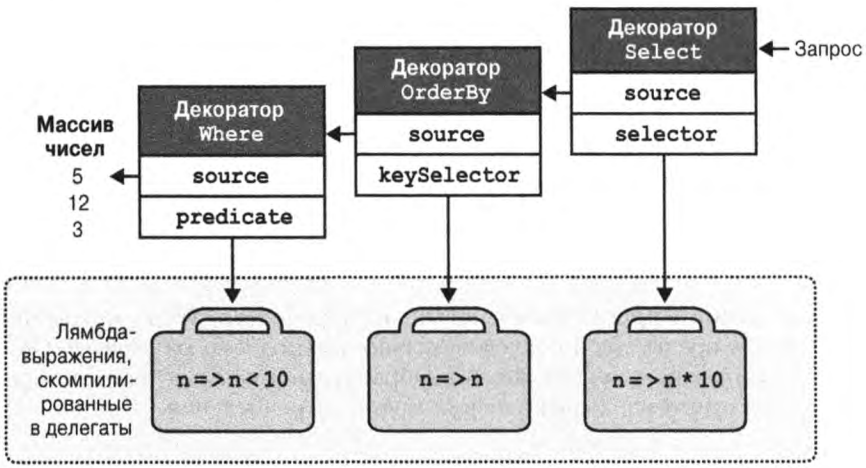



*Рис. 8.5. UML-диаграмма композиции декораторов*

## Каким образам выполняются запросы

Ниже представлены результаты перечисления предыдущего запроса:

```
foreach (int n in query) Console.WriteLine (n);
```

```
30
50
```

"За кулисами" цикл `foreach` вызывает метод `GetEnumerator` на декораторе `Select` (последняя или самая внешняя операция), который все и запускает. Результатом будет цепочка перечислителей, структурно отражающих цепочку декораторных последовательностей. На рис. 8.6 показан поток выполнения при прохождении перечисления.

В первом разделе настоящей главы мы изображали запрос как производственную линию с конвейерными лентами. Распространяя такую аналогию дальше, можно сказать, что запрос LINQ – это "ленивая" производственная линия, в которой конвейерные ленты перемещают элементы только по *требованию*. Построение запроса конструирует производственную линию со всеми составными частями на своих местах, но в остановленном состоянии. Когда потребитель запрашивает элемент (выполняет перечисление запроса), активизируется самая правая конвейерная лента; такое действие в свою очередь запускает остальные конвейерные ленты – когда требуются элементы входной последовательности.

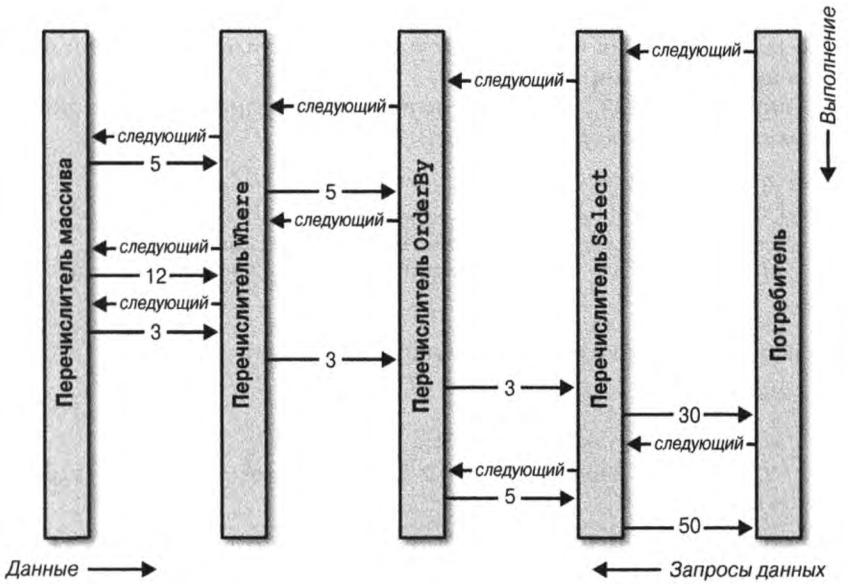



***Рис. 8.6.** Выполнение локального запроса*

Язык LINQ следует модели *с пассивным источником*, управляемой запросом, а не модели *с активным источником*, управляемой подачей. Это важный аспект, который, как будет показано далее, позволяет распространить LINQ на выдачу запросов к базам данных SQL.

# Подзапросы

Подзапрос представляет собой запрос, содержащийся внутри лямбда-выражения другого запроса. В следующем примере подзапрос применяется для сортировки музыкантов по фамилии:

```
string[] musos =
  { "David Gilmour", "Roger Waters", "Rick Wright", "Nick Mason" };

IEnumerable<string> query = musos.OrderBy (m => m.Split().Last());
```

Вызов `m.Split` преобразует каждую строку в коллекцию слов, на которой затем вызывается операция запроса `Last`. Здесь `m.Split().Last()` является подзапросом, а `query` — *внешним запросом*.

Подзапросы разрешены, т.к. с правой стороны лямбда-выражения можно помещать любое допустимое выражение C#. Подзапрос — это просто еще одно выражение C#. Таким образом, правила для подзапросов будут следствием правил для лямбда-выражений (и общего поведения операций запросов).

В общем смысле термин *подзапрос* имеет более широкое значение. При описании LINQ мы используем данный термин только для запроса, находящегося внутри лямбда-выражения другого запроса. В выражении запроса подзапрос означает запрос, на который производится ссылка из выражения в любой конструкции кроме `from`.

Подзапрос имеет закрытую область видимости внутри включающего выражения и способен ссылаться на параметры во внешнем лямбда-выражении (или переменные диапазона в выражении запроса).

Конструкция `m.Split().Last` — очень простой подзапрос. Следующий запрос извлекает из массива самые короткие строки:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> outerQuery = names
  .Where (n => n.Length == names.OrderBy (n2 => n2.Length)
                                .Select  (n2 => n2.Length).First());

Tom, Jay
```

А вот как получить то же самое с помощью выражения запроса:

```
IEnumerable<string> outerQuery =
  from   n in names
  where  n.Length ==
         (from n2 in names orderby n2.Length select n2.Length).First()
  select n;
```

Поскольку внешняя переменная диапазона (`n`) находится в области видимости подзапроса, применять `n` в качестве переменной диапазона подзапроса нельзя.

Подзапрос выполняется каждый раз, когда вычисляется включающее его лямбда-выражение. Это значит, что подзапрос выполняется по требованию, на усмотрение внешнего запроса. Можно было бы сказать, что выполнение продвигается *снаружи*. Локальные запросы следуют такой модели буквально, а интерпретируемые запросы (например, запросы к базе данных) следуют ей *концептуально*.

Подзапрос выполняется, когда это требуется для передачи данных внешнему запросу. В рассматриваемом примере подзапрос (верхняя конвейерная лента на рис. 8.7) выполняется один раз для каждой итерации внешнего цикла. Соответствующие иллюстрации приведены на рис. 8.7 и 8.8.

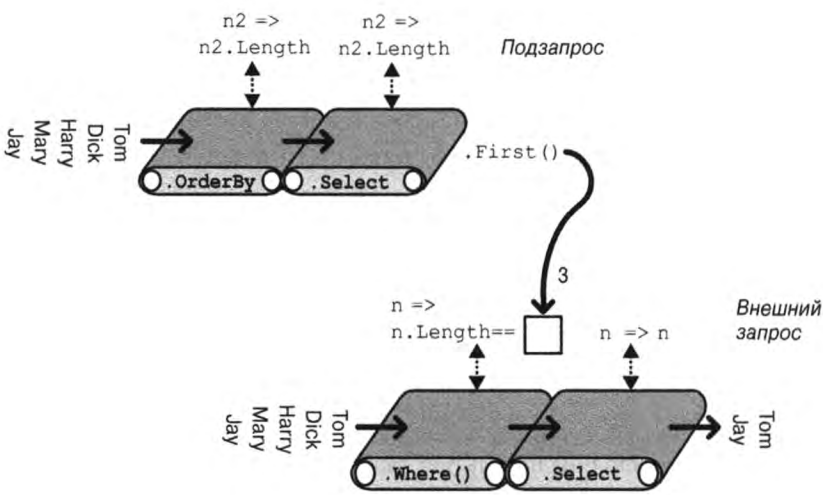



*Рис. 8.7. Композиция подзапроса*

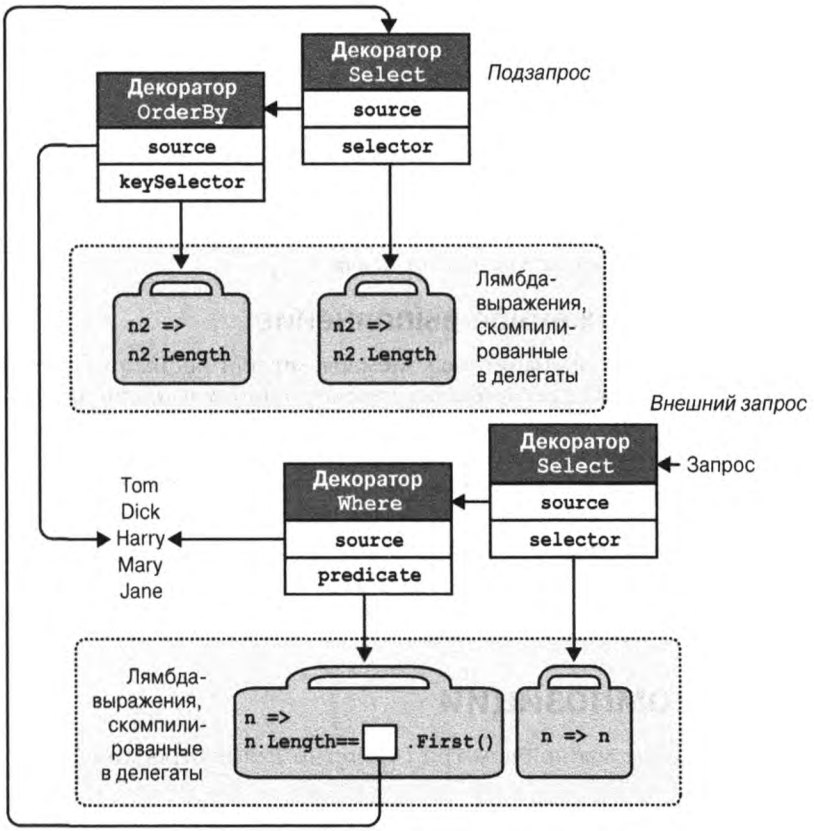



***Рис. 8.8.** UML-диаграмма композиции подзапроса*

Предыдущий подзапрос можно выразить более лаконично:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.OrderBy (n2 => n2.Length).First().Length
  select n;
```

С помощью функции агрегирования `Min` запрос можно еще больше упростить:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.Min (n2 => n2.Length)
  select n;
```

В разделе "Интерпретируемые запросы" далее в главе мы покажем, как отправлять запросы к удаленным источникам данных, таким как таблицы SQL. В нашем примере делается идеальный запрос к базе данных, потому что он может быть обработан как единое целое, требуя только одного обращения к серверу базы данных. Однако этот запрос неэффективен для локальной коллекции, т.к. на каждой итерации внешнего цикла подзапрос вычисляется повторно. Подобной неэффективности можно избежать, запуская подзапрос отдельно (так что он перестает быть подзапросом):

```
int shortest = names.Min (n => n.Length);
IEnumerable<string> query = from   n in names
                            where  n.Length == shortest
                            select n;
```

Вынесение подзапросов в подобном стиле почти всегда желательно при выполнении запросов к локальным коллекциям. Исключение – ситуация, когда подзапрос является *коррелированным*, т.е. ссылается на внешнюю переменную диапазона. Коррелированные подзапросы рассматриваются в разделе "Выполнение проекции" главы 9.

## Подзапросы и отложенное выполнение

Наличие в подзапросе операции над элементами или операции агрегирования, такой как `First` или `Count`, не приводит к немедленному выполнению *внешнего* запроса – для внешнего запроса по-прежнему поддерживается отложенное выполнение. Причина в том, что подзапросы вызываются *косвенно* – через делегат в случае локального запроса или через дерево выражения в случае интерпретируемого запроса.

Интересная ситуация возникает при помещении подзапроса внутрь выражения `Select`. Для локального запроса фактически *производится проекция последовательности запросов*, каждый из которых подпадает под отложенное выполнение. Результат обычно прозрачен и служит для дальнейшего улучшения эффективности. Подзапросы `Select` еще будут рассматриваться в главе 9.

# Стратегии композиции

В настоящем разделе мы опишем три стратегии для построения более сложных запросов:

- постепенное построение запросов;
- использование ключевого слова `into`;
- упаковка запросов.

Все они являются стратегиями *объединения в цепочки* и во время выполнения выдают идентичные запросы.

## Постепенное построение запросов

В начале главы мы демонстрировали, что текучий запрос можно было бы строить постепенно:

```
var filtered = names    .Where   (n => n.Contains ("a"));
var sorted   = filtered .OrderBy (n => n);
var query    = sorted   .Select  (n => n.ToUpper());
```

Из-за того, что каждая участвующая операция запроса возвращает декораторную последовательность, результирующим запросом будет та же самая цепочка или иерархия декораторов, которая была бы получена из запроса с единственным выражением. Тем не менее, постепенное построение запросов обладает парой потенциальных преимуществ.

- Оно может упростить написание запросов.
- Операции запросов можно добавлять *условно*. Например:

```
if (includeFilter) query = query.Where (...)
```

Это более эффективно, чем такой вариант:

```
query = query.Where (n => !includeFilter || <выражение>)
```

поскольку позволяет избежать добавления дополнительной операции запроса, если `includeFilter` равно `false`.

Постепенный подход часто полезен для понимания запросов. В целях иллюстрации предположим, что нужно удалить все гласные из списка имен и затем представить в алфавитном порядке те из них, длина которых все еще превышает два символа. С помощью текучего синтаксиса мы могли бы записать такой запрос в форме единственного выражения, произведя проецирование *перед* фильтрацией:

```
IEnumerable<string> query = names
  .Select  (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);

РЕЗУЛЬТАТ: { "Dck", "Hrry", "Mry" }
```

Вместо того чтобы вызывать метод `Replace` типа `string` пять раз, мы могли бы удалить гласные из строки более эффективно посредством регулярного выражения:

```
n => Regex.Replace (n, "[aeiou]", "")
```

Однако метод `Replace` типа `string` обладает тем преимуществом, что работает также в запросах к базам данных.

Трансляция такого кода напрямую в выражение запроса довольно непроста, потому что конструкция `select` должна находиться после конструкций `where` и `orderby`. И если переупорядочить запрос так, чтобы проецирование выполнялось последним, то результат будет другим:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length > 2
  orderby n
  select  n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
           .Replace ("o", "").Replace ("u", "");

РЕЗУЛЬТАТ: { "Dck", "Hrry", "Jy", "Mry", "Tm" }
```

К счастью, существует несколько способов получить первоначальный результат с помощью синтаксиса запросов. Первый из них — постепенное формирование запроса:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;

РЕЗУЛЬТАТ: { "Dck", "Hrry", "Mry" }
```

## Ключевое слово `into`

В зависимости от контекста ключевое слово `into` интерпретируется выражениями запросов двумя совершенно разными путями. Его первое предназначение, которое мы опишем здесь — сигнализация о *продолжении запроса* (другим предназначением является сигнализация о `GroupJoin`).

Ключевое слово into позволяет "продолжить" запрос после проецирования и является сокращением для постепенного построения запросов. Предыдущий запрос можно переписать с применением into:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "")
  into noVowel
    where noVowel.Length > 2 orderby noVowel select noVowel;
```

Единственное место, где можно использовать into – после конструкции select или group. Ключевое слово into "начинает заново" запрос, позволяя вводить новые конструкции where, orderby и select.

Хотя с точки зрения выражения запроса ключевое слово into проще считать средством начать запрос заново, после трансляции в финальную текучую форму все становится *одним запросом*. Следовательно, ключевое слово into не привносит никаких дополнительных расходов в плане производительности. Применяя его, вы совершенно ничего не теряете!

Эквивалентом into в текучем синтаксисе является просто более длинная цепочка операций.

### Правила области видимости

После ключевого слова into все переменные диапазона выходят из области видимости. Следующий код не скомпилируется:

```
var query =
  from n1 in names
  select n1.ToUpper()
  into n2                    // Начиная с этого места, видна только переменная n2
    where n1.Contains ("x") // Недопустимо: n1 не находится в области видимости
    select n2;
```

Чтобы понять причину, давайте посмотрим, как показанный код отображается на текучий синтаксис:

```
var query = names
  .Select (n1 => n1.ToUpper())
  .Where  (n2 => n1.Contains ("x"));     // Ошибка: переменная n1 не находится
                                         // в области видимости
```

К тому времени, когда запускается фильтр Where, исходная переменная (n1) уже утрачена. Входная последовательность Where содержит только имена в верхнем регистре, и ее фильтрация на основе n1 невозможна.

## Упаковка запросов

Запрос, построенный постепенно, может быть сформулирован как единственный оператор за счет упаковки одного запроса в другой. В общем случае запрос:

```
var tempQuery = tempQueryExpr
var finalQuery = from ... in tempQuery ...
```

можно переформулировать следующим образом:

```
var finalQuery = from ... in (tempQueryExpr)
```

Упаковка семантически идентична постепенному построению запросов либо использованию ключевого слова into (без промежуточной переменной). Во всех случаях конечным результатом будет линейная цепочка операций запросов. Например, рассмотрим показанный ниже запрос:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;
```

Вот как он выглядит, когда переведен в упакованную форму:

```
IEnumerable<string> query =
  from n1 in
  (
    from   n2 in names
    select n2.Replace ("a", "").Replace ("e", "").Replace ("i", "")
            .Replace ("o", "").Replace ("u", "")
  )
  where n1.Length > 2 orderby n1 select n1;
```

После преобразования в текучий синтаксис в результате получается та же самая линейная цепочка операций, что и в предшествующих примерах:

```
IEnumerable<string> query = names
  .Select  (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);
```

(Компилятор не выдает финальный вызов .Select (n => n), т.к. он избыточный.)

Упакованные запросы могут несколько запутывать, поскольку они имеют сходство с *подзапросами*, которые рассматривались ранее. Обе разновидности поддерживают концепцию внутреннего и внешнего запроса. Однако при преобразовании в текучий синтаксис можно заметить, что упаковка – это просто стратегия для последовательного выстраивания операций в цепочку. Конечный результат совершенно не похож на подзапрос, который встраивает внутренний запрос в *лямбда-выражение* другого запроса.

Возвращаясь к ранее примененной аналогии: при упаковке "внутренний" запрос имитирует *предшествующую конвейерную ленту*. И напротив, подзапрос перемещается по конвейерной ленте и активизируется по требованию посредством "лямбда-рабочего" конвейерной ленты (см. рис. 8.7).

# Стратегии проекции

## Инициализаторы объектов

До сих пор все наши конструкции select проецировали в скалярные типы элементов. С помощью инициализаторов объектов C# можно выполнять проецирование в более сложные типы. Например, предположим, что в качестве первого шага запроса мы хотим удалить гласные из списка имен, одновременно сохраняя рядом исходные версии для последующих запросов. В помощь этому мы можем написать следующий класс:

```
class TempProjectionItem
{
  public string Original;   // Исходное имя
  public string Vowelless;  // Имя с удаленными гласными
}
```

и затем проецировать в него посредством инициализаторов объектов:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<TempProjectionItem> temp =
  from n in names
  select new TempProjectionItem
  {
    Original  = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  };
```

Результатом будет тип `IEnumerable<TempProjectionItem>`, которому впоследствии можно отправлять запросы:

```
IEnumerable<string> query = from   item in temp
                            where  item.Vowelless.Length > 2
                            select item.Original;

Dick
Harry
Mary
```

## Анонимные типы

Анонимные типы позволяют структурировать промежуточные результаты без написания специальных классов. С помощью анонимных типов в предыдущем примере можно избавиться от класса `TempProjectionItem`:

```
var intermediate = from n in names
  select new
  {
    Original = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  };

IEnumerable<string> query = from   item in intermediate
                            where  item.Vowelless.Length > 2
                            select item.Original;
```

Результат будет таким же, как в предыдущем примере, но без необходимости в написании одноразового класса. Всю нужную работу проделает компилятор, сгенерировав класс с полями, которые соответствуют структуре нашей проекции. Тем не менее, это означает, что запрос `intermediate` имеет следующий тип:

```
IEnumerable <случайное-имя-сгенерированное-компилятором>
```

Единственный способ объявления переменной такого типа предусматривает использование ключевого слова `var`. В данном случае `var` является не просто средством сокращения беспорядка, а настоятельной необходимостью.

С применением ключевого слова `into` запрос можно записать более лаконично:

```
var query = from n in names
  select new
  {
    Original = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  }
  into temp
  where temp.Vowelless.Length > 2
  select temp.Original;
```

Выражения запросов предлагают сокращение для написания запросов подобного вида – ключевое слово `let`.

## Ключевое слово `let`

Ключевое слово `let` вводит новую переменную параллельно переменной диапазона.

Написать запрос, извлекающий строки, длина которых после исключения гласных превышает два символа, посредством `let` можно так:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query =
  from n in names
  let vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                   .Replace ("o", "").Replace ("u", "")
  where vowelless.Length > 2
  orderby vowelless
  select n; //Благодаря let переменная n по-прежнему находится в области видимости
```

Компилятор распознает конструкцию `let` путем проецирования во временный анонимный тип, который содержит переменную диапазона и новую переменную выражения. Другими словами, компилятор транслирует этот запрос в показанный ранее пример. Ключевое слово `let` решает две задачи:

- оно проецирует новые элементы наряду с существующими элементами;
- оно позволяет выражению многократно использоваться в запросе без необходимости в переписывании.

В приведенном примере подход с `let` особенно полезен, т.к. он позволяет конструкции `select` выполнять проецирование либо исходного имени (`n`), либо его версии с удаленными гласными (`vowelless`).

Можно иметь любое количество конструкций `let`, находящихся до или после `where` (см. рис. 8.2). В операторе `let` можно ссылаться на переменные, введенные в более ранних операторах `let` (в зависимости от границ, наложенных конструкцией `into`). Оператор `let` *повторно проецирует* все существующие переменные прозрачным образом.

Выражение `let` не должно вычисляться как значение скалярного типа: его иногда полезно оценивать, скажем, в подпоследовательность.

# Интерпретируемые запросы

Язык LINQ параллельно поддерживает две архитектуры: *локальные* запросы для локальных коллекций объектов и *интерпретируемые* запросы для удаленных источников

данных. До сих пор мы исследовали архитектуру локальных запросов, которые действуют на коллекциях, реализующих интерфейс IEnumerable<T>. Локальные запросы преобразуются в операции запросов из класса Enumerable (по умолчанию), которые в свою очередь распознаются как цепочки декораторных последовательностей. Делегаты, которые они принимают – выраженные с применением синтаксиса запросов, текучего синтаксиса или же традиционные делегаты – полностью локальны по отношению к коду на промежуточном языке (Intermediate Language – IL), в точности как любой другой метод C#.

В противоположность этому интерпретируемые запросы являются *дескриптивными*. Они действуют на последовательностях, реализующих интерфейс IQueryable<T>, и преобразуются в операции запросов из класса Queryable, которые выпускают *деревья выражений*, интерпретируемые во время выполнения.

Операции запросов в классе Enumerable могут в действительности работать с последовательностями IQueryable<T>. Сложность в том, что результирующие запросы всегда выполняются локально на стороне клиента – именно потому в классе Queryable предлагается второй набор операций запросов.

В .NET Framework доступны две реализации IQueryable<T>:

- LINQ to SQL
- Entity Framework (EF)

Указанные технологии применения LINQ для баз данных по своей поддержке в LINQ очень похожи: запросы LINQ для баз данных в книге будут работать как с LINQ to SQL, так и с EF, если только не указано обратное.

Вызывая метод AsQueryable, можно также сгенерировать оболочку IQueryable<T> для обычной перечислимой коллекции. Мы опишем метод AsQueryable в разделе "Построение выражений запросов" далее в главе.

В настоящем разделе для иллюстрации архитектуры интерпретируемых запросов мы будем использовать LINQ to SQL, потому что LINQ to SQL позволяет производить запрос без предварительного создания *модели сущностных данных* (Entity Data Model – EDM). Тем не менее, запросы, которые мы напишем, будут работать одинаково хорошо и с инфраструктурой Entity Framework (а также многими продуктами третьих сторон).

Интерфейс IQueryable<T> является расширением интерфейса IEnumerable<T> с дополнительными методами, предназначенными для конструирования деревьев выражений. Большую часть времени вы будете игнорировать детали, связанные с этими методами; они вызываются инфраструктурой .NET Framework косвенно. Интерфейс IQueryable<T> более подробно рассматривается в разделе "Построение выражений запросов" далее в главе.

Предположим, что мы создали в SQL Server простую таблицу заказчиков (Customer) и наполнили ее несколькими именами с применением следующего SQL-сценария:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30)
)
```

```
insert Customer values (1, 'Tom')
insert Customer values (2, 'Dick')
insert Customer values (3, 'Harry')
insert Customer values (4, 'Mary')
insert Customer values (5, 'Jay')
```

Располагая такой таблицей, мы можем написать на C# интерпретируемый запрос LINQ для извлечения заказчиков, имена которых содержат букву "a":

```
using System;
using System.Linq;
using System.Data.Linq;  // в сборке System.Data.Linq.dll
using System.Data.Linq.Mapping;

[Table] public class Customer
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string Name;
}
class Test
{
  static void Main()
  {
    DataContext dataContext = new DataContext ("строка подключения");
    Table<Customer> customers = dataContext.GetTable <Customer>();

    IQueryable<string> query = from c in customers
      where   c.Name.Contains ("a")
      orderby c.Name.Length
      select  c.Name.ToUpper();

    foreach (string name in query) Console.WriteLine (name);
  }
}
```

Инфраструктура LINQ to SQL транслирует запрос в следующий SQL-оператор:

```
SELECT UPPER([t0].[Name]) AS [value]
FROM [Customer] AS [t0]
WHERE [t0].[Name] LIKE @p0
ORDER BY LEN([t0].[Name])
```

Конечный результат выглядит так:

```
JAY
MARY
HARRY
```

## Каким образом работают интерпретируемые запросы

Давайте посмотрим, как обрабатывается показанный ранее запрос.

Сначала компилятор преобразует синтаксис запросов в текучий синтаксис. Это делается в точности так, как с локальными запросами:

```
IQueryable<string> query = customers.Where   (n => n.Name.Contains ("a"))
                                     .OrderBy (n => n.Name.Length)
                                     .Select  (n => n.Name.ToUpper());
```

Далее компилятор распознает методы операций запросов. Здесь локальные и интерпретируемые запросы отличаются – интерпретируемые запросы преобразуются в операции запросов из класса `Queryable`, а не класса `Enumerable`.

Чтобы понять причину, необходимо взглянуть на переменную `customers`, являющуюся источником, на котором строится весь запрос. Переменная `customers` имеет тип `Table<T>`, реализующий интерфейс `IQueryable<T>` (подтип `IEnumerable<T>`). Другими словами, у компилятора имеется выбор при распознавании `Where`: он может вызвать расширяющий метод в `Enumerable` или следующий расширяющий метод в `Queryable`:

```
public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression <Func<TSource,bool>> predicate)
```

Компилятор выбирает метод `Queryable.Where`, потому что его сигнатура обеспечивает *более специфичное соответствие*.

Метод `Queryable.Where` принимает предикат, помещенный в оболочку типа `Expression<TDelegate>`. Тем самым компилятору сообщается о необходимости транслировать переданное лямбда-выражение, т.е. `n=>n.Name.Contains("a")`, в дерево выражения, а не в компилируемый делегат. Дерево выражения — это объектная модель, основанная на типах из пространства имен `System.Linq.Expressions`, которая может инспектироваться во время выполнения (так что инфраструктура LINQ to SQL или EF может позже транслировать ее в SQL-оператор). Поскольку метод `Queryable.Where` также возвращает экземпляр реализации `IQueryable<T>`, для операций `OrderBy` и `Select` происходит аналогичный процесс. Конечный результат показан на рис. 8.9. В затененном прямоугольнике представлено *дерево выражения*, описывающее полный запрос, которое можно обходить во время выполнения.

## Выполнение

Интерпретируемые запросы следуют модели отложенного выполнения — подобно локальным запросам. Это означает, что SQL-оператор не генерируется вплоть до начала перечисления запроса. Кроме того, двукратное перечисление одного и того же запроса приводит к двум отправкам запроса в базу данных.

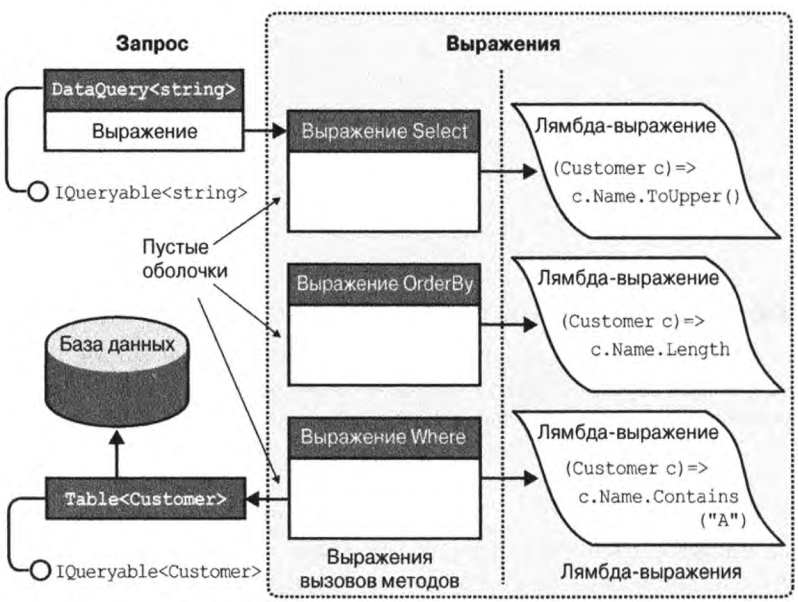



***Рис. 8.9. Композиция интерпретируемого запроса***

Внутренне интерпретируемые запросы отличаются от локальных запросов тем, как они выполняются. При перечислении интерпретируемого запроса самая внешняя последовательность запускает программу, которая обходит все дерево выражения, обрабатывая его как единое целое. В нашем примере инфраструктура LINQ to SQL транслирует дерево выражения в SQL-оператор, который затем выполняется, выдавая результаты в виде последовательности.

Для работы LINQ to SQL необходима такая информация, как схема базы данных. Именно этой цели служат атрибуты `Table` и `Column`, примененные к классу `Customer`. Упомянутые атрибуты подробно рассматриваются в разделе "LINQ to SQL и Entity Framework" далее в главе. В Entity Framework все аналогично, но требуется также модель EDM – XML-файл, в котором описано отображение между базой данных и сущностями.

Ранее уже упоминалось, что запрос LINQ подобен производственной линии. Однако выполнение перечисления конвейерной ленты `IQueryable` не приводит к запуску всей производственной линии, как в случае локального запроса. Взамен запускается только конвейерная лента `IQueryable` со специальным перечислителем, который вызывается на диспетчере производственной линии.

Диспетчер просматривает целую конвейерную ленту, которая состоит не из скомпилированного кода, а из *заглушек* (выражений вызовов методов) с инструкциями, вставленными в начале (деревья выражений). Затем диспетчер обходит все выражения, в данном случае переписывая их в единую сущность (SQL-оператор), которая затем выполняется, выдавая результаты обратно потребителю. Запускается только одна конвейерная лента; остаток производственной линии представляет собой сеть из пустых ячеек, существующих только для описания того, что должно быть сделано.

Из сказанного вытекает несколько практических последствий. Например, для локальных запросов можно записывать собственные методы запросов (довольно просто с помощью итераторов) и затем использовать их для дополнения предопределенного набора.

В отношении удаленных запросов это трудно и даже нежелательно. Если вы написали расширяющий метод `MyWhere`, принимающий `IQueryable<T>`, то хотели бы помещать собственную заглушку в производственную линию. Диспетчеру производственной линии неизвестно, что делать с вашей заглушкой. Даже если бы вы вмешались в данную фазу, то решением была бы жесткая привязка к конкретному поставщику, такому как LINQ to SQL, и невозможность работы с другими реализациями интерфейса `IQueryable`.

Одно из преимуществ наличия в `Queryable` стандартного набора методов заключается в том, что они определяют *стандартный словарь* для выдачи запросов к *любой* удаленной коллекции. Попытка расширения такого словаря приведет к утрате возможности взаимодействия.

Из модели вытекает еще одно следствие: поставщик `IQueryable` может оказаться не в состоянии справиться с некоторыми запросами – даже если вы придерживаетесь стандартных методов. Инфраструктуры LINQ to SQL и EF ограничены возможностями сервера базы данных; для некоторых запросов LINQ не предусмотрена трансляция в SQL. Если вы знакомы с языком SQL, тогда имеете об этом хорошее представление, хотя иногда приходится экспериментировать, чтобы посмотреть, что вызвало ошибку во время выполнения. Вас может удивить, что некоторые средства *вообще* работают!

## Комбинирование интерпретируемых и локальных запросов

Запрос может включать и интерпретируемые, и локальные операции. В типичном шаблоне применяются локальные операции *снаружи* и интерпретируемые компоненты *внутри*; другими словами, интерпретируемые запросы наполняют локальные запросы. Такой шаблон хорошо работает с запросами LINQ к базам данных.

Например, предположим, что мы написали специальный расширяющий метод для объединения в пары строк из коллекции:

```
public static IEnumerable<string> Pair (this IEnumerable<string> source)
{
  string firstHalf = null;
  foreach (string element in source)
    if (firstHalf == null)
      firstHalf = element;
    else
    {
      yield return firstHalf + ", " + element;
      firstHalf = null;
    }
}
```

Этот расширяющий метод можно использовать в запросе со смесью операций LINQ to SQL и локальных операций:

```
DataContext dataContext = new DataContext ("строка подключения");
Table<Customer> customers = dataContext.GetTable <Customer>();

IEnumerable<string> q = customers
  .Select  (c => c.Name.ToUpper())
  .OrderBy (n => n)
  .Pair()  // Локальные, начиная с этой точки
  .Select ((n, i) => "Pair " + i.ToString() + " = " + n);

foreach (string element in q) Console.WriteLine (element);

Pair 0 = HARRY, MARY
Pair 1 = TOM, DICK
```

Поскольку `customers` имеет тип, реализующий интерфейс `IQueryable<T>`, операция `Select` преобразуется в вызов метода `Queryable.Select`. Данный метод возвращает выходную последовательность, также имеющую тип `IQueryable<T>`, поэтому операция `OrderBy` аналогичным образом преобразуется в вызов метода `Queryable.OrderBy`. Но следующая операция запроса, `Pair`, не имеет перегруженной версии, которая принимала бы тип `IQueryable<T>` — есть только версия, принимающая менее специфичный тип `IEnumerable<T>`. Таким образом, она преобразуется в наш локальный метод `Pair` с упаковкой интерпретируемого запроса в локальный запрос. Метод `Pair` также возвращает реализацию интерфейса `IEnumerable`, а потому последующая операция `Select` преобразуется в еще одну локальную операцию.

На стороне LINQ to SQL результирующий SQL-оператор эквивалентен такому коду:

```
SELECT UPPER (Name) FROM Customer ORDER BY UPPER (Name)
```

Оставшаяся часть работы выполняется локально. Фактически мы получаем локальный запрос (снаружи), источником которого является интерпретируемый запрос (внутри).

## AsEnumerable

Метод Enumerable.AsEnumerable — простейшая из всех операций запросов. Вот его полное определение:

```
public static IEnumerable<TSource> AsEnumerable<TSource>
        (this IEnumerable<TSource> source)
{
   return source;
}
```

Он предназначен для приведения последовательности IQueryable<T> к IEnumerable<T>, заставляя последующие операции запросов привязываться к операциям из класса Enumerable вместо их привязки к операциям из класса Queryable. Это приводит к тому, что остаток запроса выполняется локально.

В целях иллюстрации предположим, что в базе данных SQL Server имеется таблица MedicalArticles, и с помощью LINQ to SQL или EF необходимо извлечь все статьи, посвященные гриппу (influenza), резюме которых содержит менее 100 слов. Для последнего предиката потребуется регулярное выражение:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

var query = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza" &&
                     wordCounter.Matches (article.Abstract).Count < 100);
```

Проблема в том, что база данных SQL Server не поддерживает регулярные выражения, поэтому поставщики LINQ для баз данных будут генерировать исключение, сообщая о невозможности трансляции запроса в SQL. Мы можем решить проблему за два шага: сначала извлечь все статьи, посвященные гриппу, посредством запроса LINQ to SQL, а затем *локально* отфильтровать сопровождающие статьи резюме, которые содержат менее 100 слов:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

IEnumerable<MedicalArticle> sqlQuery = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza");

IEnumerable<MedicalArticle> localQuery = sqlQuery
  .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Так как sqlQuery имеет тип IEnumerable<MedicalArticle>, второй запрос привязывается к локальным операциям запросов, обеспечивая выполнение части фильтрации на стороне клиента.

С помощью AsEnumerable то же самое можно сделать в единственном запросе:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

var query = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza")

  .AsEnumerable()
  .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Альтернативой вызову AsEnumerable является вызов метода ToArray или ToList. Преимущество операции AsEnumerable в том, что она не приводит к немедленному выполнению запроса и не создает какой-либо структуры для хранения.

Перенос обработки запросов из сервера базы данных на сторону клиента может нанести ущерб производительности, особенно если обработка предусматривает извлечение дополнительных строк. Более эффективный (хотя и более сложный) способ решения предполагает применение интеграции CLR с SQL для открытия доступа к функции в базе данных, которая реализует регулярное выражение.

В главе 10 мы приведем дополнительные примеры использования комбинированных интерпретируемых и локальных запросов.

# LINQ to SQL и Entity Framework

В этой и следующей главах для демонстрации интерпретируемых запросов мы повсеместно применяем инфраструктуры LINQ to SQL (L2S) и Entity Framework (EF). Давайте выясним ключевые возможности упомянутых технологий.

Если вы уже знакомы с L2S, тогда можете сразу взглянуть на табл. 8.1 (в конце раздела), где приведена сводка по отличиям API-интерфейсов с точки зрения запросов.

## Сравнение LINQ to SQL и Entity Framework

Инфраструктуры LINQ to SQL и Entity Framework являются средствами объектно-реляционного отображения, поддерживающими язык LINQ. Главное отличие между ними заключается в том, что EF обеспечивает более строгое разделение между схемой базы данных и запрашиваемыми классами. Вместо запрашивания классов, которые близко представляют схему базы данных, вы имеете дело с высокоуровневой абстракцией, описываемой с помощью EDM. Это приводит к увеличению гибкости, но за счет снижения производительности и возрастания сложности.

Инфраструктура L2S была написана командой разработчиков компилятора C# и вышла в составе версии .NET Framework 3.5; инфраструктура EF была создана командой разработчиков ADO.NET и выпущена позже как часть пакета обновлений Service Pack 1. С тех пор инфраструктура L2S перешла под контроль команды разработчиков ADO.NET. В продукт были внесены лишь незначительные улучшения, после чего команда разработчиков сосредоточила основные усилия на EF.

В последних версиях инфраструктура EF была заметно усовершенствована, хотя каждая технология по-прежнему обладает уникальными достоинствами. Достоинства L2S касаются легкости использования, простоты, высокой производительности и качества трансляции в код SQL. Достоинство EF заключается в гибкости создания сложных отображений между базой данных и сущностными классами. Технология EF также позволяет работать с базами данных, отличающимися от SQL Server, через *модель поставщиков* (L2S тоже содержит в себе модель поставщиков, но она была сделана внутренней, чтобы стимулировать переход независимых разработчиков на EF).

Технология L2S является великолепным пособием для изучения того, как выполнять запросы к базам данных в LINQ, потому что аспекты, связанные с объектно-реляционным отображением, сохранены простыми, а принципы формирования запросов также применимы к EF.

## Сущностные классы LINQ to SQL

Инфраструктура L2S позволяет использовать для представления данных любой класс при условии, что он декорирован соответствующими атрибутами. Ниже приведен простой пример:

```
[Table]
public class Customer
{
  [Column(IsPrimaryKey=true)]
  public int ID;

  [Column]
  public string Name;
}
```

Атрибут `[Table]` из пространства имен `System.Data.Linq.Mapping` сообщает инфраструктуре L2S о том, что объект этого типа представляет строку в таблице базы данных. По умолчанию предполагается, что имя таблицы совпадает с именем класса; в противном случае понадобится указать имя таблицы:

```
[Table (Name="Customers")]
```

Класс, декорированный атрибутом `[Table]`, в терминологии L2S называется *сущностью*. Чтобы быть полезным, его структура должна близко — или точно — соответствовать структуре таблицы базы данных, превращая такой класс в низкоуровневую конструкцию.

Атрибутом `[Column]` помечается поле или свойство, которое отображается на столбец в таблице. Если имя столбца отличается от имени поля или свойства, тогда имя столбца можно указать так:

```
[Column (Name="FullName")]
public string Name;
```

Свойство `IsPrimaryKey` в атрибуте `[Column]` указывает на то, что столбец входит в состав первичного ключа таблицы и требуется для поддержки объектной идентичности, а также делает возможным запись обновлений обратно в базу данных.

Вместо определения открытых полей можно определить открытые свойства в сочетании с закрытыми полями. Это позволит реализовать в средствах доступа к свойствам логику проверки достоверности. Если пойти таким путем, то можно дополнительно проинструктировать L2S о необходимости пропускать средства доступа к свойствам и производить запись напрямую в поля при их наполнении из базы данных:

```
string _name;

[Column (Storage="_name")]
public string Name { get { return _name; } set { _name = value; } }
```

Конструкция `Column(Storage="_name")` сообщает инфраструктуре L2S о том, что при наполнении сущности должна производиться запись напрямую в поле `_name` (а не в свойство `Name`). Применение рефлексии в L2S позволяет полю быть закрытым, как продемонстрировано выше в примере.

Сущностные классы можно генерировать автоматически из базы данных, используя либо среду Visual Studio (для чего добавить новый элемент проекта LINQ to SQL Classes (Классы LINQ to SQL)), либо инструмент командной строки *SqlMetal*.

## Сущностные классы Entity Framework

Как и L2S, инфраструктура EF позволяет применять для представления данных любой класс (хотя потребуется реализовать специальные интерфейсы, если нужна функциональность, подобная навигационным свойствам).

Например, следующий сущностный класс представляет заказчика, который в конечном итоге отображается на таблицу *заказчиков* в базе данных:

```
// Необходима ссылка на сборку System.Data.Entity.dll
[EdmEntityType (NamespaceName = "NutshellModel", Name = "Customer")]
public partial class Customer
{
  [EdmScalarPropertyAttribute (EntityKeyProperty=true, IsNullable=false)]
  public int ID { get; set; }
  [EdmScalarProperty (EntityKeyProperty = false, IsNullable = false)]
  public string Name { get; set; }
}
```

Однако в отличие от L2S одного такого класса недостаточно. Вспомните, что в EF база данных напрямую не запрашивается, а запрос направляется высокоуровневой модели под названием EDM. Должен быть какой-нибудь способ описания модели EDM, и чаще всего он задействует XML-файл с расширением *.edmx*, состоящий из трех частей:

- *концептуальная модель*, которая описывает EDM в изоляции от базы данных;
- *модель хранения*, которая описывает схему базы данных;
- *отображение*, которое описывает, каким образом концептуальная модель отображается на хранилище.

Простейший способ создания файла *.edmx* предусматривает добавление элемента проекта ADO.NET Entity Data Model (Модель сущностных данных ADO.NET) в Visual Studio и следование указаниям мастера для генерации сущностей из базы данных. В результате создается не только файл *.edmx*, но и сущностные классы.

Сущностные классы в EF отображаются на *концептуальную модель*. Типы, которые поддерживают запрашивание и обновление концептуальной модели, вместе называются *объектными службами* (Object Services).

Визуальный конструктор предполагает, что изначально требуется простое однозначное отображение между таблицами и сущностями. Тем не менее, результат можно усовершенствовать за счет настройки модели EDM либо с помощью визуального конструктора, либо путем редактирования созданного конструктором файла *.edmx*. Ниже перечислены действия, которые вы можете предпринять:

- отобразить несколько таблиц на одну сущность;
- отобразить одну таблицу на несколько сущностей;
- отобразить унаследованные типы с использованием трех стандартных стратегий, популярных в мире ORM (object-relational mapping – объектно-реляционное отображение).

Ниже описаны три вида стратегий наследования.

**Таблица на иерархию**

Одиночная таблица отображается на целую иерархию классов. Эта таблица содержит дискриминантный столбец для указания, на какой тип должна отображаться каждая строка.

**Таблица на тип**

Одиночная таблица отображается на один тип, т.е. унаследованный тип отображается на несколько таблиц. При запросе сущности инфраструктура EF генерирует SQL-оператор `JOIN` для слияния вместе всех базовых типов.

**Таблица на конкретный тип**

На каждый конкретный тип отображается отдельная таблица. Это означает, что базовый тип отображается на несколько таблиц, и при запросе сущностей базового типа EF генерирует SQL-оператор `UNION`.

(Напротив, инфраструктура L2S поддерживает только стратегию "таблица на иерархию".)

Модель EDM отличается высокой сложностью: ее подробное описание могло бы занять несколько сотен страниц! По данной теме доступна хорошая книга Джулии Лерман под названием *Programming Entity Framework* (`http://oreilly.com/catalog/9780596520298/`).

Инфраструктура EF также позволяет запрашивать посредством модели EDM без LINQ – с применением текстового языка Entity SQL (ESQL). Такая возможность может быть удобной для динамически конструируемых запросов.

## `DataContext` и `ObjectContext`

После определения сущностных классов (и модели EDM в случае EF) можно приступать к формированию запросов. Первый шаг – создание экземпляра класса `DataContext` (L2S) или `ObjectContext` (EF) с передачей ему строки подключения:

```
var l2sContext = new DataContext ("строка подключения к базе данных");
var efContext = new ObjectContext ("строка подключения к сущности");
```

Создание экземпляра класса `DataContext/ObjectContext` напрямую представляет собой низкоуровневый подход, который удачно демонстрирует работу классов. Однако обычно создается экземпляр *типизированного контекста* (подкласса упомянутых классов), и вскоре мы рассмотрим такой процесс.

В случае L2S передается *строка подключения к базе данных*, а в случае EF должна передаваться *строка подключения к сущности*, которая содержит строку подключения к базе данных, а также информацию о том, как найти модель EDM. (Если вы создали EDM в Visual Studio, тогда ищите строку подключения к сущности для EDM в файле *app.config*.)

Затем можно получить запрашиваемый объект, вызвав метод `GetTable` (L2S) или `CreateObjectSet` (EF). В следующем примере используется класс `Customer`, который был определен ранее:

```
var context = new DataContext ("строка подключения к базе данных");
Table<Customer> customers = context.GetTable <Customer>();
Console.WriteLine (customers.Count());          // Количество строк в таблице
Customer cust = customers.Single (c => c.ID == 2); // Извлекает заказчика
                                                   // с идентификатором 2
```

Ниже приведен тот же пример, но с применением EF:

```
var context = new ObjectContext ("строка подключения к сущности");
context.DefaultContainerName = "NutshellEntities";
ObjectSet<Customer> customers = context.CreateObjectSet<Customer>();
Console.WriteLine (customers.Count());          // Количество строк в таблице
Customer cust = customers.Single (c => c.ID == 2); // Извлекает заказчика
                                                   // с идентификатором 2
```

Операция `Single` идеально подходит для извлечения строки по первичному ключу. В отличие от `First` в случае возвращения более одного элемента она генерирует исключение.

Объект `DataContext`/`ObjectContext` решает две задачи. Во-первых, он действует в качестве фабрики для генерации объектов, которые можно запрашивать. Во-вторых, он отслеживает любые изменения, внесенные в сущности, так что становится возможной их запись обратно в базу данных. Предыдущий пример можно продолжить обновлением информации о заказчике для L2S:

```
Customer cust = customers.OrderBy (c => c.Name).First();
cust.Name = "Updated Name";
context.SubmitChanges();
```

Для EF единственное отличие заключается в вызове метода `SaveChanges`:

```
Customer cust = customers.OrderBy (c => c.Name).First();
cust.Name = "Updated Name";
context.SaveChanges();
```

### Типизированные контексты

Необходимость в постоянном вызове методов `GetTable<Customer>` или `CreateObjectSet<Customer>` порождает неудобства. Более эффективный подход предполагает создание подкласса `DataContext`/`ObjectContext` для отдельной базы данных с добавлением свойств, которые делают это для каждой сущности. Результат называется *типизированным контекстом*:

```
class NutshellContext : DataContext  // Для LINQ to SQL
{
  public Table<Customer> Customers => GetTable<Customer>();
  // ... и т.д. для каждой таблицы в базе данных
}
```

А вот то же самое для EF:

```
class NutshellContext : ObjectContext  // Для Entity Framework
{
  public ObjectSet<Customer> Customers => CreateObjectSet<Customer>();
  // ... и т.д. для каждой сущности в концептуальной модели
}
```

Затем можно поступать следующим образом:

```
var context = new NutshellContext ("строка подключения");
Console.WriteLine (context.Customers.Count());
```

В случае использования Visual Studio для создания элементов проекта LINQ to SQL Classes (Классы LINQ to SQL) или ADO.NET Entity Data Model (Модель сущностных данных ADO.NET) типизированный контекст строится автоматически. Визуальные конструкторы могут также выполнять дополнительную работу, такую как применение формы множественного числа к идентификаторам – в приведенном примере использовался бы идентификатор `context.Customers`, а не `context.Customer`, невзирая на то, что таблица SQL и сущностный класс называются `Customer`.

## Освобождение экземпляров DataContext/ObjectContext

Несмотря на то что классы `DataContext/ObjectContext` реализуют интерфейс `IDisposable`, можно (в общем случае) обойтись без освобождения их экземпляров. Освобождение принудительно закрывает подключение контекста, но обычно это необязательно, т.к. инфраструктуры L2S и EF закрывают подключения автоматически всякий раз, когда завершается извлечение результатов из запроса.

Освобождение контекста в действительности может оказаться проблематичным из-за ленивой оценки. Взгляните на следующий код:

```
IQueryable<Customer> GetCustomers (string prefix)
{
  using (var dc = new NutshellContext ("строка подключения"))
    return dc.GetTable<Customer>()
             .Where (c => c.Name.StartsWith (prefix));
}
...
foreach (Customer c in GetCustomers ("a"))
  Console.WriteLine (c.Name);
```

Такой код потерпит неудачу, потому что запрос оценивается при его перечислении, которое происходит *после* освобождения экземпляра `DataContext`.

Тем не менее, ниже указаны некоторые предостережения, о которых следует помнить в случае, если контексты не освобождаются.

- Как правило, объект подключения должен освобождать все неуправляемые ресурсы в методе `Close`. В то время как сказанное справедливо для класса `SqlConnection`, объекты подключений третьих сторон теоретически могут удерживать ресурсы открытыми, если метод `Close` вызывался, а метод `Dispose` – нет (хотя такой подход, вероятно, нарушит контракт, определенный методом `IDbConnection.Close`).
- Если вы вручную вызываете метод `GetEnumerator` на запросе (вместо применения оператора `foreach`) и затем забываете либо освободить перечислитель, либо воспользоваться последовательностью, то подключение останется открытым. Освобождение экземпляра `DataContext/ObjectContext` предоставляет страховку для таких сценариев.
- Некоторые разработчики считают гораздо более аккуратным подходом освобождение контекстов (и всех объектов, которые реализуют интерфейс `IDisposable`).

Чтобы освободить контексты явно, потребуется передать экземпляр `DataContext/ObjectContext` в методы вроде `GetCustomers` и избежать описанных выше проблем.

### Отслеживание объектов

Экземпляр `DataContext/ObjectContext` отслеживает все созданные им сущности, поэтому он может поставлять одинаковые сущности каждый раз, когда запрашиваются те же самые строки таблицы. Другими словами, в течение своего существования контекст никогда не выдаст две отдельные сущности, которые ссылаются на одну и ту же строку в таблице (где строка идентифицируется первичным ключом).

Отключить указанное поведение инфраструктуры L2S можно путем установки в `false` свойства `ObjectTrackingEnabled` объекта `DataContext`. В EF отслеживание изменений можно отключать на основе типов:

```
context.Customers.MergeOption = MergeOption.NoTracking;
```

Отключение отслеживания объектов также предотвращает отправку обновлений данных.

В целях иллюстрации отслеживания объектов предположим, что заказчик, имя которого по алфавиту расположено первым, также имеет наименьший идентификатор. В следующем примере `a` и `b` будут ссылаться на один и тот же объект:

```
var context = new NutshellContext ("строка подключения");

Customer a = context.Customers.OrderBy (c => c.Name).First();
Customer b = context.Customers.OrderBy (c => c.ID).First();
```

Отсюда следует пара интересных выводов. Для начала посмотрим, что происходит, когда инфраструктура L2S или EF сталкивается со вторым запросом. Она начинает с отправки запроса базе данных и в ответ получает одиночную строку. Затем инфраструктура читает первичный ключ полученной строки и производит его поиск в кеше сущностей экземпляра контекста. Обнаружив совпадение, она возвращает существующий объект *без обновления любых значений*. Таким образом, если другой пользователь только что обновил имя текущего заказчика в базе данных, то новое значение будет проигнорировано. Это важно для устранения нежелательных побочных эффектов (объект `Customer` может использоваться где-то в другом месте) и также для управления параллелизмом. Если вы изменили свойства объекта `Customer` и пока еще не вызвали метод `SubmitChanges/SaveChanges`, то определенно не хотите, чтобы данные были автоматически перезаписаны.

Чтобы получить актуальную информацию из базы данных, потребуется либо создать новый экземпляр контекста, либо вызвать его метод `Refresh`, передав ему сущность или сущности, которые должны быть обновлены.

Второй вывод касается того, что явно проецировать в сущностный тип для выбора подмножества столбцов строки не получится без возникновения проблем. Например, если необходимо извлечь только имя заказчика, тогда допустим любой из показанных далее подходов:

```
customers.Select (c => c.Name);
customers.Select (c => new { Name = c.Name } );
customers.Select (c => new MyCustomType { Name = c.Name } );
```

Однако следующий подход не разрешен:

```
customers.Select (c => new Customer { Name = c.Name } );
```

Причина в том, что сущности Customer в конечном итоге будут заполнены только частично. Таким образом, если впоследствии выполнить запрос, который охватывает *все* столбцы записи заказчика, то будут получены кешированные объекты Customer с единственным заполненным свойством Name.

В многоуровневом приложении нельзя применять одиночный статический экземпляр DataContext или ObjectContext на промежуточном уровне для обработки всех запросов, т.к. контексты не являются безопасными в отношении потоков. Взамен методы промежуточного уровня должны создавать новый контекст для каждого клиентского запроса. На самом деле это выгодно, т.к. затраты на обработку одновременных обновлений перекладываются на сервер базы данных, который соответствующим образом оснащен для выполнения такой работы. Сервер базы данных, например, будет использовать семантику уровня изоляции транзакций.

## Ассоциации

Инструменты генерации сущностных классов выполняют еще одну полезную работу. Для каждого отношения, определенного в базе данных, на каждой его стороне они генерируют свойства, которые позволяют запрашивать такие отношения. Например, предположим, что для таблиц заказчиков и покупок определено отношение "один ко многим":

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)
create table Purchase
(
  ID int not null primary key,
  CustomerID int references Customer (ID),
  Description varchar(30) not null,
  Price decimal not null
)
```

Благодаря автоматически сгенерированным сущностным классам мы можем записывать запросы вроде показанных ниже:

```
var context = new NutshellContext ("строка подключения");

// Извлечь все покупки, сделанные первым заказчиком (в алфавитном порядке):

Customer cust1 = context.Customers.OrderBy (c => c.Name).First();
foreach (Purchase p in cust1.Purchases)
  Console.WriteLine (p.Price);

// Извлечь заказчика, совершившего покупки на минимальную сумму:

Purchase cheapest = context.Purchases.OrderBy (p => p.Price).First();
Customer cust2 = cheapest.Customer;
```

Кроме того, если окажется, что экземпляры cust1 и cust2 ссылаются на одного и того же заказчика, то c1 и c2 будут *ссылаться на один и тот же объект*: cust1==cust2 в результате даст true.

Давайте исследуем сигнатуру автоматически сгенерированного свойства Purchases в сущностном классе Customer. В случае L2S это свойство выглядит так:

```
[Association (Storage="_Purchases", OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases { get {...} set {...} }
```

А вот каким оно будет в EF:

```
[EdmRelationshipNavigationProperty ("NutshellModel", "FK...", "Purchase")]
public EntityCollection<Purchase> Purchases { get {...} set {...} }
```

Свойство `EntitySet` или `EntityCollection` похоже на предопределенный запрос со встроенной конструкцией `Where`, которая извлекает связанные сущности. Атрибут `[Association]` предоставляет инфраструктуре L2S информацию, необходимую для формирования SQL-оператора; атрибут `[EdmRelationshipNavigationProperty]` сообщает инфраструктуре EF, где в модели EDM искать сведения об этом отношении.

Как и с другими типами запросов, выполнение является отложенным. В случае L2S свойство `EntitySet` наполняется при перечислении; в случае EF свойство `EntityCollection` наполняется при явном вызове метода `Load`. Ниже показано свойство `Purchases.Customer` на другой стороне отношения для L2S:

```
[Association (Storage="_Customer",ThisKey="CustomerID",IsForeignKey=true)]
public Customer Customer { get {...} set {...} }
```

Хотя свойство имеет тип `Customer`, лежащее в основе поле (`_Customer`) относится к типу `EntityRef`. Тип `EntityRef` реализует отложенную загрузку, так что связанная сущность `Customer` не извлекается из базы данных до тех пор, пока она не будет действительно затребована.

Инфраструктура EF работает аналогично за исключением того, что не наполняет свойство просто при обращении к нему: наполнение требует вызова метода `Load` на объекте `EntityReference`. Это значит, что контексты EF должны открывать доступ к свойствам как для действительного родительского объекта, так и для его оболочки `EntityReference`:

```
[EdmRelationshipNavigationProperty ("NutshellModel", "FK..., "Customer")]
public Customer Customer { get {...} set {...} }

public EntityReference<Customer> CustomerReference { get; set; }
```

Поведение инфраструктуры EF можно сделать похожим на поведение L2S, заставив ее наполнять коллекции `EntityCollection` и `EntityReference` просто путем доступа к их свойствам:

```
context.ContextOptions.DeferredLoadingEnabled = true;
```

## Отложенное выполнение в L2S и EF

Запросы L2S и EF подчиняются отложенному выполнению в точности как локальные запросы. Это позволяет строить запросы постепенно. Тем не менее, существует один аспект, в котором инфраструктура L2S/EF поддерживает специальную семантику отложенного выполнения, и возникает он в ситуации, когда подзапрос находится внутри выражения `Select`.

- В локальных запросах получается двойное отложенное выполнение, поскольку с функциональной точки зрения осуществляется выборка последовательности *запросов*. Таким образом, если перечислять внешнюю результирующую последовательность, но не перечислять внутренние последовательности, то подзапрос никогда не выполнится.
- В L2S/EF подзапрос выполняется в то же самое время, когда выполняется главный внешний запрос. В итоге появляется возможность избежать излишних обращений к серверу.

Например, следующий запрос выполняется в одиночном обращении при достижении первого оператора foreach:

```
var context = new NutshellContext ("строка подключения");

var query = from c in context.Customers
            select
              from p in c.Purchases
              select new { c.Name, p.Price };

foreach (var customerPurchaseResults in query)
  foreach (var namePrice in customerPurchaseResults)
    Console.WriteLine (namePrice.Name + " spent " + namePrice.Price);
```

Любые свойства EntitySet/EntityCollection, которые явно спроецированы, полностью наполняются в единственном обращении к серверу:

```
var query = from c in context.Customers
            select new { c.Name, c.Purchases };

foreach (var row in query)
  foreach (Purchase p in row.Purchases) // Дополнительные обращения
                                        // к серверу отсутствуют
    Console.WriteLine (row.Name + " spent " + p.Price);
```

Но если проводить перечисление свойств EntitySet/EntityCollection без первоначального проецирования, то будут применяться правила отложенного выполнения. В приведенном ниже примере инфраструктуры L2S и EF выполняют еще один запрос свойства Purchases на каждой итерации цикла:

```
context.ContextOptions.DeferredLoadingEnabled = true;  // Только для EF

foreach (Customer c in context.Customers)
  foreach (Purchase p in c.Purchases)   // Еще одно обращение к серверу
    Console.WriteLine (c.Name + " spent " + p.Price);
```

Такая модель полезна, когда нужно выполнять внутренний цикл *избирательно*, основываясь на проверке, которая может быть сделана только на стороне клиента:

```
foreach (Customer c in context.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
    foreach (Purchase p in c.Purchases)  // Еще одно обращение к серверу
      Console.WriteLine (...);
```

(Подзапросы Select более детально исследуются в разделе "Выполнение проекции" главы 9.)

Как видите, за счет явного проецирования ассоциаций можно избежать лишних обращений к серверу. Инфраструктуры L2S и EF предлагают для этого также и другие механизмы, которые мы рассмотрим в следующих двух разделах.

## DataLoadOptions

Класс DataLoadOptions специфичен для инфраструктуры L2S. С ним связаны два разных сценария использования:

- он позволяет заблаговременно указывать фильтр для ассоциаций EntitySet (AssociateWith);
- он позволяет запрашивать энергичную загрузку определенных EntitySet для сокращения количества обращений к серверу (LoadWith).

### Заблаговременное указание фильтра

Давайте перепишем предыдущий пример следующим образом:

```
foreach (Customer c in context.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
    ProcessCustomer (c);
```

Мы определим метод ProcessCustomer так, как показано ниже:

```
void ProcessCustomer (Customer c)
{
  Console.WriteLine (c.ID + " " + c.Name);
  foreach (Purchase p in c.Purchases)
    Console.WriteLine ("  - purchased a " + p.Description);
}
```

Теперь предположим, что методу ProcessCustomer необходимо передавать только *подмножество* покупок для каждого заказчика — скажем, самых дорогостоящих. Ниже продемонстрировано одно из решений:

```
foreach (Customer c in context.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
    ProcessCustomer (c.ID,
                     c.Name,
                     c.Purchases.Where (p => p.Price > 1000));
...
void ProcessCustomer (int custID, string custName,
                      IEnumerable<Purchase> purchases)
{
  Console.WriteLine (custID + " " + custName);
  foreach (Purchase p in purchases)
    Console.WriteLine ("  - purchased a " + p.Description);
}
```

Код выглядит запутанным. Он станет еще более запутанным, когда методу ProcessCustomer понадобится передавать больше полей Customer. Более удачное решение предусматривает применение метода AssociateWith класса DataLoadOptions:

```
DataLoadOptions options = new DataLoadOptions();
options.AssociateWith <Customer>
  (c => c.Purchases.Where (p => p.Price > 1000));
context.LoadOptions = options;
```

Здесь экземпляр DataContext инструктируется о том, что свойство Purchases в Customer нужно всегда фильтровать, используя заданный предикат. Теперь мы можем работать с исходной версией метода ProcessCustomer.

Метод AssociateWith не изменяет семантику отложенного выполнения. Когда применяется конкретное отношение, оно просто указывает на необходимость неявного добавления дополнительного фильтра.

### Энергичная загрузка

Второй сценарий использования класса DataLoadOptions связан с требованием для определенных экземпляров EntitySet энергичной загрузки вместе с их родительской сущностью. Например, предположим, что необходимо загрузить всех заказчиков и их покупки в рамках единственного обращения к серверу. Именно это делает следующий код:

```
DataLoadOptions options = new DataLoadOptions();
options.LoadWith <Customer> (c => c.Purchases);
context.LoadOptions = options;

foreach (Customer c in context.Customers)  // Одно обращение к серверу:
  foreach (Purchase p in c.Purchases)
    Console.WriteLine (c.Name + " bought a " + p.Description);
```

Здесь указывается, что при каждом извлечении сущности `Customer` одновременно должна извлекаться связанная с ней сущность `Purchases`. Метод `LoadWith` можно комбинировать с `AssociateWith`. Приведенный далее код обеспечивает извлечение информации о *самых дорогостоящих* покупках во время извлечения заказчика в том же самом обращении к серверу:

```
options.LoadWith <Customer> (c => c.Purchases);
options.AssociateWith <Customer>
  (c => c.Purchases.Where (p => p.Price > 1000));
```

## Энергичная загрузка в Entity Framework

Затребовать энергичную загрузку ассоциаций в EF можно с помощью метода `Include`. Следующий код выполняет перечисление покупок каждого заказчика, одновременно генерируя только один SQL-запрос:

```
foreach (Customer c in context.Customers.Include ("Purchases"))
  foreach (Purchase p in c.Purchases)
    Console.WriteLine (p.Description);
```

Метод `Include` может применяться произвольно широко и глубоко. Например, если каждая сущность `Purchase` также имеет навигационные свойства `PurchaseDetails` и `SalesPersons`, тогда организовать энергичную загрузку всей вложенной структуры можно так:

```
context.Customers.Include ("Purchases.PurchaseDetails")
                 .Include ("Purchases.SalesPersons")
```

## Обновления

Инфраструктуры L2S и EF также отслеживают изменения, вносимые в сущности, и позволяют записывать их обратно в базу данных путем вызова метода `SubmitChanges` на объекте `DataContext` или метода `SaveChanges` на объекте `ObjectContext`.

Класс `Table<T>` в L2S предоставляет методы `InsertOnSubmit` и `DeleteOnSubmit`, предназначенные для вставки и удаления строк в таблице, а класс `ObjectSet<T>` в EF для тех же целей предлагает методы `AddObject` и `DeleteObject`. Вот как вставить строку:

```
var context = new NutshellContext ("строка подключения");

Customer cust = new Customer { ID=1000, Name="Bloggs" };
context.Customers.InsertOnSubmit (cust);    // AddObject в случае EF
context.SubmitChanges();                    // SaveChanges в случае EF
```

Позже вставленную строку можно извлечь, обновить и, наконец, удалить:

```
var context = new NutshellContext ("строка подключения");

Customer cust = context.Customers.Single (c => c.ID == 1000);
cust.Name = "Bloggs2";
context.SubmitChanges();                    // Обновляет сведения о заказчике

context.Customers.DeleteOnSubmit (cust);    // DeleteObject в случае EF
context.SubmitChanges();                    // Удаляет сведения о заказчике
```

Метод SubmitChanges/SaveChanges собирает все изменения, которые были внесены в сущности с момента создания (или последнего сохранения) экземпляра контекста, и затем выполняет SQL-оператор для их записи в базу данных. При формировании транзакции принимается во внимание любой указанный экземпляр класса TransactionScope; в его отсутствие все операторы помещаются в новую транзакцию.

В EntitySet/EntityCollection можно также добавлять новые или существующие строки, вызывая метод Add. При этом инфраструктуры L2S и EF автоматически заполняют внешние ключи (после вызова метода SubmitChanges или SaveChanges):

```
Purchase p1 = new Purchase { ID=100, Description="Bike",  Price=500 };
Purchase p2 = new Purchase { ID=101, Description="Tools", Price=100 };

Customer cust = context.Customers.Single (c => c.ID == 1);

cust.Purchases.Add (p1);
cust.Purchases.Add (p2);

context.SubmitChanges();  // SaveChanges в случае EF
```

Если вы не хотите иметь дело с накладными расходами, связанными с выделением уникальных ключей, то можете использовать для первичного ключа либо автоинкрементное поле (IDENTITY в SQL Server), либо Guid.

В следующем примере инфраструктура L2S/EF автоматически помещает 1 в столбец CustomerID каждой новой записи о покупках (инфраструктуре L2S известно об этом благодаря атрибуту ассоциации, определенному на свойстве Purchases, а EF получает информацию из модели EDM):

```
[Association (Storage="_Purchases", OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases { get {...} set {...} }
```

Если сущностные классы Customer и Purchase были сгенерированы визуальным конструктором Visual Studio или инструментом командной строки SqlMetal, то эти классы будут включать дополнительный код для поддержания двух сторон каждого отношения в синхронизированном состоянии. Другими словами, присваивание свойства Purchase.Customer приведет к автоматическому добавлению нового заказчика в набор сущностей Customer.Purchases — и наоборот. Сказанное можно проиллюстрировать, переписав предыдущий пример следующим образом:

```
var context = new NutshellContext ("строка подключения");

Customer cust = context.Customers.Single (c => c.ID == 1);
new Purchase { ID=100, Description="Bike",  Price=500, Customer=cust };
new Purchase { ID=101, Description="Tools", Price=100, Customer=cust };

context.SubmitChanges();  // SaveChanges в случае EF
```

Когда строка удаляется из EntitySet/EntityCollection, ее поле внешнего ключа устанавливается в null. В следующем коде разрывается ассоциация между последними двумя добавленными покупками и заказчиком:

```
var context = new NutshellContext ("строка подключения");

Customer cust = context.Customers.Single (c => c.ID == 1);

cust.Purchases.Remove (cust.Purchases.Single (p => p.ID == 100));
cust.Purchases.Remove (cust.Purchases.Single (p => p.ID == 101));

context.SubmitChanges()  // Отправить SQL-оператор
                         // в базу данных (SaveChanges в случае EF)
```

Поскольку при этом производится попытка установить поле CustomerID каждой записи о покупке в null, столбец Purchase.CustomerID в базе данных должен допускать значения null; иначе сгенерируется исключение. (Кроме того, поле CustomerID или свойство в сущностном классе должно принадлежать типу, допускающему null.)

Чтобы удалить все дочерние сущности, удалите их из Table<T> или ObjectSet<T> (это означает, что сначала их придется извлечь). В случае L2S работа выполняется так:

```
var c = context;
c.Purchases.DeleteOnSubmit (c.Purchases.Single (p => p.ID == 100));
c.Purchases.DeleteOnSubmit (c.Purchases.Single (p => p.ID == 101));
c.SubmitChanges();  // Отправить SQL-оператор в базу данных
```

А в случае EF следующим образом:

```
var c = context;
c.Purchases.DeleteObject (c.Purchases.Single (p => p.ID == 100));
c.Purchases.DeleteObject (c.Purchases.Single (p => p.ID == 101));
c.SaveChanges();    // Отправить SQL-оператор в базу данных
```

## Отличия между API-интерфейсами L2S и EF

Как было показано выше, инфраструктуры L2S и EF похожи в плане выдачи запросов с помощью LINQ и выполнения обновлений. В табл. 8.1 представлены отличия между их API-интерфейсами.

**Таблица 8.1. Отличия между API-интерфейсами L2S и EF**

| Средство | LINQ to SQL | Entity Framework |
|---|---|---|
| Управляющий класс для всех операций CRUD (create, read, update, delete — создание, чтение, обновление, удаление) | DataContext | ObjectContext |
| Метод для (ленивого) извлечения всех сущностей заданного типа из хранилища | GetTable | CreateObjectSet |
| Тип, возвращаемый предыдущим методом | Table<T> | ObjectSet<T> |
| Метод для обновления хранилища любыми добавлениями, модификациями или удалениями сущностных объектов | SubmitChanges | SaveChanges |
| Метод для добавления новой сущности, когда контекст обновляется | InsertOnSubmit | AddObject |
| Метод для удаления сущности из хранилища, когда контекст обновляется | DeleteOnSubmit | DeleteObject |
| Тип для представления одной стороны свойства отношения, когда эта сторона является множественной | EntitySet<T> | EntityCollection<T> |
| Тип для представления одной стороны свойства отношения, когда эта сторона является одиночной | EntityRef<T> | EntityReference<T> |
| Стандартная стратегия для загрузки свойств отношений | Ленивая | Явная |
| Конструкция, которая включает энергичную загрузку | DataLoadOptions | .Include() |

# Построение выражений запросов

Когда возникала необходимость в динамически сформированных запросах, ранее в главе мы условно соединяли в цепочки операции запросов. Хотя такой прием вполне адекватен для многих сценариев, иногда требуется работать на более детализированном уровне и динамически составлять лямбда-выражения, которые передаются операциям.

В настоящем разделе мы предполагаем, что класс `Product` определен так:

```
[Table] public partial class Product
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string Description;
  [Column]                    public bool Discontinued;
  [Column]                    public DateTime LastSale;
}
```

## Сравнение делегатов и деревьев выражений

Вспомните, что:

- локальные запросы, которые используют операции `Enumerable`, принимают делегаты;
- интерпретируемые запросы, которые используют операции `Queryable`, принимают деревья выражений.

В этом легко удостовериться, сравнив сигнатуры операции `Where` в классах `Enumerable` и `Queryable`:

```
public static IEnumerable<TSource> Where<TSource> (this
  IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression<Func<TSource,bool>> predicate)
```

Лямбда-выражение, внедренное в запрос, выглядит идентично независимо от того, привязано оно к операциям класса `Enumerable` или к операциям класса `Queryable`:

```
IEnumerable<Product> q1 = localProducts.Where (p => !p.Discontinued);
IQueryable<Product>  q2 = sqlProducts.Where   (p => !p.Discontinued);
```

Однако в случае присваивания лямбда-выражения промежуточной переменной потребуется принять явное решение относительно ее преобразования в делегат (т.е. `Func<>`) или в дерево выражения (т.е. `Expression<Func<>>`). В следующем примере переменные `predicate1` и `predicate2` не являются взаимозаменяемыми:

```
Func <Product, bool> predicate1 = p => !p.Discontinued;
IEnumerable<Product> q1 = localProducts.Where (predicate1);

Expression <Func <Product, bool>> predicate2 = p => !p.Discontinued;
IQueryable<Product> q2 = sqlProducts.Where (predicate2);
```

### Компиляция деревьев выражений

Дерево выражения можно преобразовать в делегат, вызвав метод `Compile`. Прием особенно полезен при написании методов, которые возвращают многократно применяемые выражения. В целях иллюстрации мы добавим в класс `Product` статический метод, возвращающий предикат, который вычисляется в `true`, если товар не снят с производства и был продан на протяжении последних 30 дней:

```
public partial class Product
{
  public static Expression<Func<Product, bool>> IsSelling()
  {
    return p => !p.Discontinued && p.LastSale > DateTime.Now.AddDays (-30);
  }
}
```

(Мы определили метод в отдельном частичном классе во избежание перезаписывания его автоматическим генератором DataContext, таким как генератор кода Visual Studio.)

Только что написанный метод можно использовать как в интерпретируемых, так и в локальных запросах:

```
void Test()
{
  var dataContext = new NutshellContext ("строка подключения");
  Product[] localProducts = dataContext.Products.ToArray();

  IQueryable<Product> sqlQuery =
    dataContext.Products.Where (Product.IsSelling());

  IEnumerable<Product> localQuery =
    localProducts.Where (Product.IsSelling.Compile());
}
```

Инфраструктура .NET Framework не предоставляет API-интерфейса для преобразования в обратном направлении, т.е. из делегата в дерево выражения, что делает деревья выражений более универсальными.

### AsQueryable

Операция AsQueryable позволяет записывать целые *запросы*, которые могут запускаться в отношении локальных или удаленных последовательностей:

```
IQueryable<Product> FilterSortProducts (IQueryable<Product> input)
{
  return from p in input
         where ...
         order by ...
         select p;
}

void Test()
{
  var dataContext = new NutshellContext ("строка подключения");
  Product[] localProducts = dataContext.Products.ToArray();

  var sqlQuery   = FilterSortProducts (dataContext.Products);
  var localQuery = FilterSortProducts (localProducts.AsQueryable());
  ...
}
```

Операция AsQueryable помещает локальную последовательность в оболочку IQueryable<T>, так что последующие операции запросов преобразуются в деревья выражений. Если позже выполнить перечисление результата, тогда деревья выражений неявно скомпилируются (за счет небольшого снижения производительности), и локальная последовательность будет перечисляться обычным образом.

## Деревья выражений

Ранее уже упоминалось, что неявное преобразование из лямбда-выражения в класс `Expression<TDelegate>` заставляет компилятор C# генерировать код, который строит дерево выражения. Приложив небольшие усилия по программированию, то же самое можно сделать вручную во время выполнения – другими словами, динамически построить дерево выражения с нуля. Результат можно привести к типу `Expression<TDelegate>` и применять в запросах LINQ к базам данных или скомпилировать в обычный делегат, вызвав метод `Compile`.

### DOM-модель выражения

Дерево выражения – это миниатюрная кодовая DOM-модель. Каждый узел в дереве представлен типом из пространства имен `System.Linq.Expressions`, которые показаны на рис. 8.10.

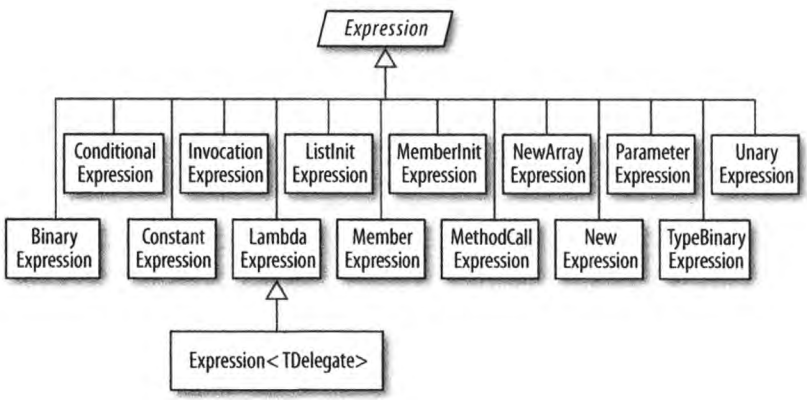



***Рис. 8.10.*** *Типы выражений*

Начиная с .NET Framework 4.0, пространство имен `System.Linq.Expressions` предлагает дополнительные типы выражений и методы для поддержки языковых конструкций, которые могут присутствовать в блоках кода. Они полезны для среды DLR, а не для лямбда-выражений. Другими словами, лямбда-выражения в стиле блоков кода по-прежнему не могут быть преобразованы в деревья выражений:

```
Expression<Func<Customer,bool>> invalid =
  c => { return true; }  // Блоки кода не разрешены
```

Базовым классом для всех узлов является (необобщенный) класс `Expression`. Обобщенный класс `Expression<TDelegate>` в действительности означает "типизированное лямбда-выражение" и мог бы называться `LambdaExpression<TDelegate>`, если бы следующая запись не выглядела настолько неуклюжей:

```
LambdaExpression<Func<Customer,bool>> f = ...
```

Базовым типом `Expression<T>` является (необобщенный) класс `LambdaExpression`. Класс `LamdbaExpression` обеспечивает унификацию типов для деревьев лямбда-вы-

ражений: любой типизированный экземпляр `Expression<T>` может быть приведен к типу `LambdaExpression`.

Отличие `LambdaExpression` от обычного класса `Expression` связано с тем, что лямбда-выражения имеют *параметры*.

Чтобы создать дерево выражения, не нужно создавать экземпляры типов узлов напрямую; взамен следует вызывать статические методы, предлагаемые классом `Expression`, список которых приведен ниже:

```
Add                    GreaterThan            NegateChecked
AddChecked             GreaterThanOrEqual     New
And                    Invoke                 NewArrayBounds
AndAlso                Lambda                 NewArrayInit
ArrayIndex             LeftShift              Not
ArrayLength            LessThan               NotEqual
Bind                   LessThanOrEqual        Or
Call                   ListBind               OrElse
Coalesce               ListInit               Parameter
Condition              MakeBinary             Power
Constant               MakeMemberAccess       Property
Convert                MakeUnary              PropertyOrField
ConvertChecked         MemberBind             Quote
Divide                 MemberInit             RightShift
ElementInit            Modulo                 Subtract
Equal                  Multiply               SubtractChecked
ExclusiveOr            MultiplyChecked        TypeAs
Field                  Negate                 TypeIsëUnaryPlus
```

На рис. 8.11 представлено дерево выражения, которое создается в результате следующего присваивания:

```
Expression<Func<string, bool>> f = s => s.Length < 5;
```

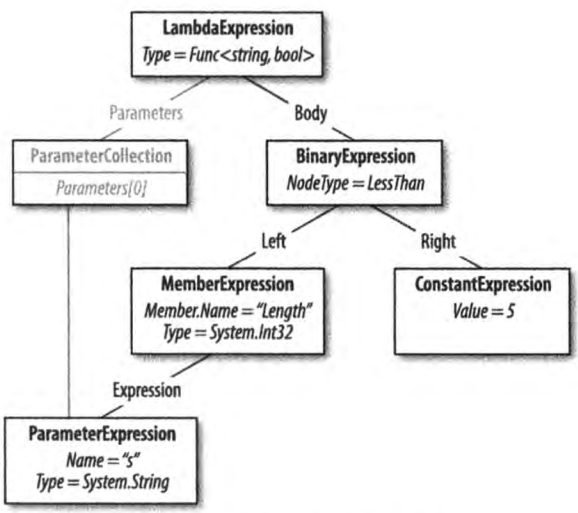



***Рис. 8.11.** Дерево выражения*

Это можно продемонстрировать с помощью такого кода:

```
Console.WriteLine (f.Body.NodeType);                    // LessThan
Console.WriteLine (((BinaryExpression) f.Body).Right);  // 5
```

Давайте теперь построим такое выражение с нуля. Принцип состоит в том, чтобы начать с нижней части дерева и работать, двигаясь вверх. В самой нижней части дерева находится экземпляр класса `ParameterExpression` — параметр лямбда-выражения по имени `s` типа `string`:

```
ParameterExpression p = Expression.Parameter (typeof (string), "s");
```

На следующем шаге строятся экземпляры классов `MemberExpression` и `ConstantExpression`. В первом случае необходимо получить доступ к *свойству* `Length` параметра `s`:

```
MemberExpression stringLength = Expression.Property (p, "Length");
ConstantExpression five = Expression.Constant (5);
```

Далее создается сравнение `LessThan` (меньше чем):

```
BinaryExpression comparison = Expression.LessThan (stringLength, five);
```

На финальном шаге конструируется лямбда-выражение, которое связывает свойство `Body` выражения с коллекцией параметров:

```
Expression<Func<string, bool>> lambda
  = Expression.Lambda<Func<string, bool>> (comparison, p);
```

Удобный способ тестирования построенного лямбда-выражения предусматривает его компиляцию в делегат:

```
Func<string, bool> runnable = lambda.Compile();

Console.WriteLine (runnable ("kangaroo"));      // False
Console.WriteLine (runnable ("dog"));           // True
```

Выяснить тип выражения, который должен использоваться, проще всего, просмотрев существующее лямбда-выражение в отладчике Visual Studio.

Дополнительные рассуждения по данной теме можно найти по адресу `http://www.albahari.com/expressions/`.

# 9

# Операции LINQ

В настоящей главе подробно описаны все операции запросов LINQ. Для справочных целей в разделах "Выполнение проекции" и "Выполнение соединения" далее в главе раскрывается несколько концептуально важных областей:

- проецирование иерархий объектов;
- соединение с помощью `Select`, `SelectMany`, `Join` и `GroupJoin`;
- выражения запросов с множеством переменных диапазона.

Во всех примерах главы предполагается, что массив `names` определен следующим образом:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

В примерах, которые запрашивают базу данных, экземпляр типизированной переменной `DataContext` по имени `dataContext` создается так, как показано ниже:

```
var dataContext = new NutshellContext ("строка подключения");
...
public class NutshellContext : DataContext
{
  public NutshellContext (string cxString) : base (cxString) {}
  public Table<Customer> Customers { get { return GetTable<Customer>(); } }
  public Table<Purchase> Purchases { get { return GetTable<Purchase>(); } }
}
[Table] public class Customer
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string Name;
  [Association (OtherKey="CustomerID")]
  public EntitySet<Purchase> Purchases = new EntitySet<Purchase>();
}
[Table] public class Purchase
{
    [Column(IsPrimaryKey=true)] public int ID;
    [Column]                    public int? CustomerID;
    [Column]                    public string Description;
    [Column]                    public decimal Price;
    [Column]                    public DateTime Date;
```

```
  EntityRef<Customer> custRef;
  [Association (Storage="custRef",ThisKey="CustomerID",IsForeignKey=true)]
  public Customer Customer
  {
    get { return custRef.Entity; } set { custRef.Entity = value; }
  }
}
```

Все примеры, которые рассматриваются в главе, предварительно загружены в LINQPad вместе с примером базы данных, имеющей подходящую схему. Загрузить LINQPad можно на веб-сайте `www.linqpad.net`.

Показанные сущностные классы являются упрощенной версией того, что обычно производят инструменты LINQ to SQL, и не включают код для обновления противоположной стороны отношения в случае переустановки его сущностей.

Ниже приведены соответствующие определения SQL-таблиц:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)
create table Purchase
(
  ID int not null primary key,
  CustomerID int references Customer (ID),
  Description varchar(30) not null,
  Price decimal not null
)
```

Все примеры также будут работать с инфраструктурой Entity Framework за исключением случаев, для которых указано обратное. На основе этих таблиц можно построить класс `ObjectContext` для Entity Framework, создав новую модель EDM в Visual Studio и перетащив значки таблиц на поверхность визуального конструктора.

# Обзор

В данном разделе мы представим обзор стандартных операций запросов.

Стандартные операции запросов разделены на три категории:

- последовательность на входе, последовательность на выходе (последовательность→последовательность);
- последовательность на входе, одиночный элемент или скалярное значение на выходе;
- ничего на входе, последовательность на выходе (методы *генерации*).

Сначала мы рассмотрим каждую из трех категорий и укажем, какие операции запросов она включает, а затем перейдем к детальному описанию индивидуальных операций.

## Последовательность → последовательность

В данную категорию попадает большинство операций запросов – они принимают одну или более последовательностей на входе и выпускают одиночную выходную последовательность. На рис. 9.1 показаны операции, которые реструктурируют форму последовательностей.

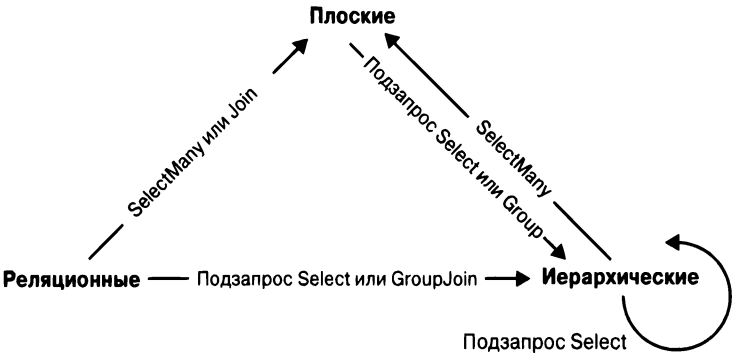



*Рис. 9.1. Операции, изменяющие форму последовательностей*

### Фильтрация

**`IEnumerable<TSource>` → `IEnumerable<TSource>`**

Возвращает подмножество исходных элементов:

`Where, Take, TakeWhile, Skip, SkipWhile, Distinct`

### Проецирование

**`IEnumerable<TSource>` → `IEnumerable<TResult>`**

Трансформирует каждый элемент с помощью лямбда-функции. Операция `SelectMany` выравнивает вложенные последовательности; операции `Select` и `SelectMany` выполняют внутренние соединения, левые внешние соединения, перекрестные соединения и неэквисоединения с помощью LINQ to SQL и EF:

`Select, SelectMany`

### Соединение

**`IEnumerable<TOuter>`, `IEnumerable<TInner>` → `IEnumerable<TResult>`**

Объединяет элементы одной последовательности с элементами другой. Операции `Join` и `GroupJoin` спроектированы для эффективной работы с локальными запросами и поддерживают внутренние и левые внешние соединения. Операция `Zip` перечисляет две последовательности за раз, применяя функцию к каждой паре элементов. Вместо имен `TOuter` и `TInner` для аргументов типов в операции `Zip` используются имена `TFirst` и `TSecond`:

```
IEnumerable<TFirst>, IEnumerable<TSecond> → IEnumerable<TResult>
Join, GroupJoin, Zip
```

### Упорядочение

**`IEnumerable<TSource>` → `IOrderedEnumerable<TSource>`**

Возвращает переупорядоченную последовательность:

`OrderBy`, `ThenBy`, `Reverse`

### Группирование

**`IEnumerable<TSource>` → `IEnumerable<IGrouping<TSource,TElement>>`**

Группирует последовательность в подпоследовательности:

`GroupBy`

### Операции над множествами

**`IEnumerable<TSource>`, `IEnumerable<TSource>` → `IEnumerable<TSource>`**

Принимает две последовательности одного и того же типа и возвращает их общность, сумму или разницу:

`Concat`, `Union`, `Intersect`, `Except`

### Методы преобразования: импортирование

**`IEnumerable` → `IEnumerable<TResult>`**

`OfType`, `Cast`

### Методы преобразования: экспортирование

**`IEnumerable<TSource>` → массив, список, словарь, объект `Lookup` или последовательность:**

`ToArray`, `ToList`, `ToDictionary`, `ToLookup`, `AsEnumerable`, `AsQueryable`

## Последовательность → элемент или значение

Описанные ниже операции запросов принимают входную последовательность и выдают одиночный элемент или значение.

### Операции над элементами

**`IEnumerable<TSource>` → `TSource`**

Выбирает одиночный элемент из последовательности:

`First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault`, `ElementAt`, `ElementAtOrDefault`, `DefaultIfEmpty`

### Методы агрегирования

**`IEnumerable<TSource>` → *скалярное значение***

Выполняет вычисление над последовательностью, возвращая скалярное значение (обычно число):

`Aggregate`, `Average`, `Count`, `LongCount`, `Sum`, `Max`, `Min`

### Квантификаторы

**`IEnumerable<TSource>` → *значение* `bool`**

Агрегация, возвращающая `true` или `false`:

`All`, `Any`, `Contains`, `SequenceEqual`

## Ничего → последовательность

К третьей и последней категории относятся операции, которые строят выходную последовательность с нуля.

### Методы генерации

**Ничего → `IEnumerable<TResult>`**

Производит простую последовательность:

`Empty, Range, Repeat`

# Выполнение фильтрации

**`IEnumerable<TSource>` → `IEnumerable<TSource>`**

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| `Where` | Возвращает подмножество элементов, удовлетворяющих заданному условию | `WHERE` |
| `Take` | Возвращает первые `count` элементов и отбрасывает остальные | `WHERE ROW_NUMBER()...` *или* подзапрос `TOP n` |
| `Skip` | Пропускает первые `count` элементов и возвращает остальные | `WHERE ROW_NUMBER()...` *или* `NOT IN (SELECT TOP n...)` |
| `TakeWhile` | Выдает элементы из входной последовательности до тех пор, пока предикат не станет равным `false` | Генерируется исключение |
| `SkipWhile` | Пропускает элементы из входной последовательности до тех пор, пока предикат не станет равным `false`, после чего возвращает остальные элементы | Генерируется исключение |
| `Distinct` | Возвращает последовательность, из которой исключены дубликаты | `SELECT DISTINCT...` |

Информация в колонке "Эквиваленты в SQL" справочных таблиц в настоящей главе не обязательно соответствует тому, что будет производить реализация интерфейса `IQueryable`, такая как LINQ to SQL. Взамен в ней показано то, что обычно применяется для выполнения той же работы при написании SQL-запроса вручную. Если простая трансляция отсутствует, тогда ячейка в этой колонке остается пустой. Если же трансляции вообще не существует, то указывается примечание "Генерируется исключение".

Код реализации в `Enumerable`, когда он приводится, не включает проверку на предмет `null` для аргументов и предикатов индексации.

Каждый метод фильтрации всегда выдает либо такое же, либо меньшее количество элементов по сравнению с начальным их числом. Получить большее количество элементов невозможно! Кроме того, на выходе получаются идентичные элементы; они никак не трансформируются.

## Where

| Аргумент | Тип |
|---|---|
| Исходная последовательность | IEnumerable<TSource> |
| Предикат | TSource => bool или (TSource,int) => bool* |

* Запрещено в случае LINQ to SQL и Entity Framework.

### Синтаксис запросов

```
where выражение-bool
```

### Реализация в Enumerable

Если оставить в стороне проверки на равенство null, то внутренняя реализация операции Enumerable.Where функционально эквивалентна следующему коду:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func <TSource, bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

### Обзор

Операция Where возвращает элементы входной последовательности, которые удовлетворяют заданному предикату.

Например:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names.Where (name => name.EndsWith ("y"));

// Результат: { "Harry", "Mary", "Jay" }
```

Или в синтаксисе запросов:

```
IEnumerable<string> query = from n in names
                            where n.EndsWith ("y")
                            select n;
```

Конструкция where может встречаться в запросе более одного раза, перемежаясь с конструкциями let, orderby и join:

```
from n in names
where n.Length > 3
let u = n.ToUpper()
where u.EndsWith ("Y")
select u;  // Результат: { "HARRY", "MARY" }
```

К таким запросам применяются стандартные правила области видимости C#. Другими словами, на переменную нельзя ссылаться до ее объявления как переменной диапазона или с помощью конструкции let.

### Индексированная фильтрация

Предикат операции Where дополнительно принимает второй аргумент типа int. В него помещается позиция каждого элемента внутри входной последовательности, позволяя предикату использовать эту информацию в своем решении по фильтрации.

Например, следующий запрос обеспечивает пропуск каждого второго элемента:

```
IEnumerable<string> query = names.Where ((n, i) => i % 2 == 0);
// Результат: { "Tom", "Harry", "Jay" }
```

Попытка применения индексированной фильтрации в LINQ to SQL или EF приводит к генерации исключения.

### Сравнения с помощью SQL-операции LIKE в LINQ to SQL и EF

Следующие методы, выполняемые на строках, транслируются в SQL-операцию LIKE:

```
Contains, StartsWith, EndsWith
```

Например, `c.Name.Contains ("abc")` транслируется в конструкцию `customer.Name LIKE '%abc%'` (точнее, в ее параметризованную версию). Метод `Contains` позволяет сравнивать только с локально вычисляемым выражением; для сравнения с другим столбцом придется использовать метод `SqlMethods.Like`:

```
... where SqlMethods.Like (c.Description, "%" + c.Name + "%")
```

Метод `SqlMethods.Like` также позволяет выполнять более сложные сравнения (скажем, `LIKE 'abc%def%'`).

### Строковые сравнения < и > в LINQ to SQL и EF

С помощью метода `CompareTo` типа `string` можно выполнять сравнение *порядка* для строк; он отображается на SQL-операции < и >:

```
dataContext.Purchases.Where (p => p.Description.CompareTo ("C") < 0)
```

### WHERE x IN (..., ..., ...) в LINQ to SQL и EF

В инфраструктурах LINQ to SQL и EF операцию `Contains` можно применять к локальной коллекции внутри предиката фильтра. Например:

```
string[] chosenOnes = { "Tom", "Jay" };

from c in dataContext.Customers
where chosenOnes.Contains (c.Name)
...
```

Это отображается на SQL-операцию `IN`, т.е.:

```
WHERE customer.Name IN ("Tom", "Jay")
```

Если локальная коллекция является массивом сущностей или элементов нескалярных типов, то инфраструктура LINQ to SQL или EF может взамен выпустить конструкцию `EXISTS`.

## Take и Skip

| Аргумент | Тип |
|---|---|
| Исходная последовательность | `IEnumerable<TSource>` |
| Количество элементов, которые необходимо выдать или пропустить | `int` |

Операция Take выдает первые *n* элементов и отбрасывает остальные; операция Skip отбрасывает первые *n* элементов и выдает остальные. Эти два метода удобно применять вместе при реализации веб-страницы, позволяющей пользователю перемещаться по крупному набору записей. Например, пусть пользователь выполняет поиск в базе данных книг термина "mercury" (ртуть) и получает 100 совпадений. Приведенный ниже запрос возвращает первые 20 найденных книг:

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Take (20);
```

Следующий запрос возвращает книги с 21-й по 40-ю:

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Skip (20).Take (20);
```

Инфраструктуры LINQ to SQL и EF транслируют операции Take и Skip в функцию ROW_NUMBER для SQL Server 2005 и выше или в подзапрос TOP *n* для предшествующих версий SQL Server.

## TakeWhile и SkipWhile

| Аргумент | Тип |
|---|---|
| Исходная последовательность | IEnumerable<TSource> |
| Предикат | TSource => bool или (TSource,int) => bool |

Операция TakeWhile выполняет перечисление входной последовательности, выдавая элементы до тех пор, пока заданный предикат не станет равным false. Оставшиеся элементы игнорируются:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var takeWhileSmall = numbers.TakeWhile (n => n < 100);  // { 3, 5, 2 }
```

Операция SkipWhile выполняет перечисление входной последовательности, пропуская элементы до тех пор, пока заданный предикат не станет равным false. Оставшиеся элементы выдаются:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var skipWhileSmall = numbers.SkipWhile (n => n < 100);  // { 234, 4, 1 }
```

Операции TakeWhile и SkipWhile не транслируются в SQL и приводят к генерации исключения, когда присутствуют в запросе LINQ к базам данных.

## Distinct

Операция Distinct возвращает входную последовательность, из которой удалены дубликаты. Дополнительно можно передавать специальный компаратор эквивалентности. Следующий запрос возвращает отличающиеся буквы в строке:

```
char[] distinctLetters = "HelloWorld".Distinct().ToArray();
string s = new string (distinctLetters);  // HeloWrd
```

Методы LINQ можно вызывать прямо на строке, т.к. тип string реализует интерфейс IEnumerable<char>.

# Выполнение проекции

**IEnumerable<TSource> → IEnumerable<TResult>**

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| Select | Трансформирует каждый входной элемент с помощью заданного лямбда-выражения | SELECT |
| SelectMany | Трансформирует каждый входной элемент, а затем выравнивает и объединяет результирующие подпоследовательности | INNER JOIN, LEFT OUTER JOIN, CROSS JOIN |

При запрашивании базы данных операции Select и SelectMany являются наиболее универсальными конструкциями соединения; для локальных запросов операции Join и GroupJoin — самые *эффективные* конструкции соединения.

## Select

| Аргумент | Тип |
|---|---|
| Исходная последовательность | IEnumerable<TSource> |
| Селектор результатов | TSource => TResult или (TSource,int) => TResult* |

* Запрещено в случае LINQ to SQL и Entity Framework.

### Синтаксис запросов

```
select выражение-проекции
```

### Реализация в Enumerable

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

### Обзор

Посредством операции Select вы всегда получаете то же самое количество элементов, с которого начинали. Однако с помощью лямбда-функции каждый элемент может быть трансформирован произвольным образом.

Следующий запрос выбирает имена всех шрифтов, установленных на компьютере (через свойство FontFamily.Families из пространства имен System.Drawing):

```
IEnumerable<string> query = from f in FontFamily.Families
                            select f.Name;
foreach (string name in query) Console.WriteLine (name);
```

В этом примере конструкция select преобразует объект FontFamily в его имя. Ниже приведен лямбда-эквивалент:

```
IEnumerable<string> query = FontFamily.Families.Select (f => f.Name);
```

Операторы `Select` часто используются для проецирования в анонимные типы:

```
var query =
  from f in FontFamily.Families
  select new { f.Name, LineSpacing = f.GetLineSpacing (FontStyle.Bold) };
```

Проекция без трансформации иногда применяется в синтаксисе запросов с целью удовлетворения требования о том, что запрос должен заканчиваться конструкцией `select` или `group`. Следующий запрос извлекает шрифты, поддерживающие зачеркивание:

```
IEnumerable<FontFamily> query =
  from f in FontFamily.Families
  where f.IsStyleAvailable (FontStyle.Strikeout)
  select f;

foreach (FontFamily ff in query) Console.WriteLine (ff.Name);
```

В таких случаях при переводе в текучий синтаксис компилятор опускает проекцию.

### Индексированное проецирование

Выражение селектора может дополнительно принимать целочисленный аргумент, который действует в качестве индексатора, предоставляя выражение с позицией каждого элемента во входной последовательности. Прием работает только с локальными запросами:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query = names
  .Select ((s,i) => i + "=" + s);    // { "0=Tom", "1=Dick", ... }
```

### Подзапросы `Select` и иерархии объектов

Для построения иерархии объектов подзапрос можно вкладывать внутрь конструкции `select`. В следующем примере возвращается коллекция, которая описывает каждый каталог в *D:\source*, с коллекцией файлов, хранящихся внутри каждого подкаталога:

```
DirectoryInfo[] dirs = new DirectoryInfo (@"d:\source").GetDirectories();

var query =
  from d in dirs
  where (d.Attributes & FileAttributes.System) == 0
  select new
  {
    DirectoryName = d.FullName,
    Created = d.CreationTime,
    Files = from f in d.GetFiles()
            where (f.Attributes & FileAttributes.Hidden) == 0
            select new { FileName = f.Name, f.Length, }
  };

foreach (var dirFiles in query)
{
  Console.WriteLine ("Directory: " + dirFiles.DirectoryName);
  foreach (var file in dirFiles.Files)
    Console.WriteLine ("  " + file.FileName + " Len: " + file.Length);
}
```

Внутреннюю часть запроса можно назвать *коррелированным подзапросом*. Подзапрос является коррелированным, если он ссылается на объект во внешнем запросе; в данном случае это d — каталог, который перечисляется.

Подзапрос внутри Select позволяет отображать одну иерархию объектов на другую или отображать реляционную объектную модель на иерархическую объектную модель.

В локальных запросах подзапрос внутри Select приводит к дважды отложенному выполнению. В приведенном выше примере файлы не будут фильтроваться или проецироваться до тех пор, пока внутренний цикл foreach не начнет перечисление.

### Подзапросы и соединения в LINQ to SQL и EF

Проекции подзапросов эффективно функционируют в инфраструктурах LINQ to SQL и EF и могут использоваться для выполнения работы соединений в стиле SQL. Ниже показано, как извлечь для каждого заказчика имя и его дорогостоящие покупки:

```
var query =
  from c in dataContext.Customers
  select new {
             c.Name,
             Purchases = from p in dataContext.Purchases
                         where p.CustomerID == c.ID && p.Price > 1000
                         select new { p.Description, p.Price }
           };
foreach (var namePurchases in query)
{
  Console.WriteLine ("Customer: " + namePurchases.Name);
  foreach (var purchaseDetail in namePurchases.Purchases)
    Console.WriteLine ("  - $$$: " + purchaseDetail.Price);
}
```

Такой стиль запроса идеально подходит для интерпретируемых запросов. Внешний запрос и подзапрос обрабатываются как единое целое, что позволяет избежать излишних обращений к серверу. Однако в случае локальных запросов это неэффективно, т.к. для получения нескольких соответствующих комбинаций потребуется выполнить перечисление каждой комбинации внешнего и внутреннего элементов. Более удачное решение для локальных запросов обеспечивают операции Join и GroupJoin, которые описаны в последующих разделах.

Приведенный выше запрос сопоставляет объекты из двух разных коллекций, и его можно считать "соединением". Отличие между ним и обычным соединением базы данных (или подзапросом) заключается в том, что вывод не выравнивается в одиночный двумерный результирующий набор. Мы отображаем реляционные данные на иерархические, а не на плоские данные.

Вот как выглядит тот же самый запрос, упрощенный за счет применения свойства ассоциации Purchases сущностного класса Customer:

```
from c in dataContext.Customers
select new
{
  c.Name,
```

```
  Purchases = from p in c.Purchases  // Purchases - это EntitySet<Purchase>
              where p.Price > 1000
              select new { p.Description, p.Price }
};
```

Оба запроса аналогичны левому внешнему соединению в SQL в том смысле, что при внешнем перечислении мы получаем всех заказчиков независимо от того, связаны с ними какие-либо покупки. Для эмуляции внутреннего соединения, при котором исключаются заказчики, не совершившие дорогостоящих покупок, необходимо добавить к коллекции покупок условие фильтрации:

```
from c in dataContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select new {
            c.Name,
            Purchases = from p in c.Purchases
                        where p.Price > 1000
                        select new { p.Description, p.Price }
          };
```

Запрос выглядит слегка неаккуратно из-за того, что один и тот же предикат (`Price > 1000`) записан дважды. Избежать подобного дублирования можно посредством конструкции `let`:

```
from c in dataContext.Customers
let highValueP = from p in c.Purchases
                 where p.Price > 1000
                 select new { p.Description, p.Price }
where highValueP.Any()
select new { c.Name, Purchases = highValueP };
```

Представленный стиль запроса отличается определенной гибкостью. Например, изменив `Any` на `Count`, мы можем модифицировать запрос с целью извлечения только заказчиков, совершивших, по меньшей мере, две дорогостоящие покупки:

```
...
where highValueP.Count() >= 2
select new { c.Name, Purchases = highValueP };
```

### Проецирование в конкретные типы

Проецирование в анонимные типы удобно при получении промежуточных результатов, но не особенно подходит, например, когда нужно отправить результирующий набор клиенту, поскольку анонимные типы могут существовать только в виде локальных переменных внутри метода. Альтернативой является использование для проекций конкретных типов, таких как `DataSet` или классы специальных бизнес-сущностей. Специальная бизнес-сущность — это просто класс, который вы разрабатываете. Подобно классу, аннотированному с помощью `[Table]` в LINQ to SQL, или сущности в EF он содержит ряд свойств, но спроектирован для сокрытия низкоуровневых деталей (касающихся базы данных). К примеру, из классов бизнес-сущностей могут быть исключены поля внешних ключей. Предполагая, что специальные сущностные классы `CustomerEntity` и `PurchaseEntity` уже написаны, ниже показано, как можно было бы выполнить проецирование в них:

```
IQueryable<CustomerEntity> query =
  from c in dataContext.Customers
  select new CustomerEntity
  {
    Name = c.Name,
```

```
    Purchases =
      (from p in c.Purchases
       where p.Price > 1000
       select new PurchaseEntity {
                                    Description = p.Description,
                                    Value = p.Price
                                  }
      ).ToList()
  };
// Обеспечить выполнение запроса, преобразовав вывод в более удобный список:
List<CustomerEntity> result = query.ToList();
```

Обратите внимание, что до сих пор мы не должны были применять операции `Join` или `SelectMany`. Причина в том, что мы поддерживали иерархическую форму данных, как показано на рис. 9.2. В LINQ часто удается избежать традиционного подхода SQL, при котором таблицы выравниваются в двухмерный результирующий набор.

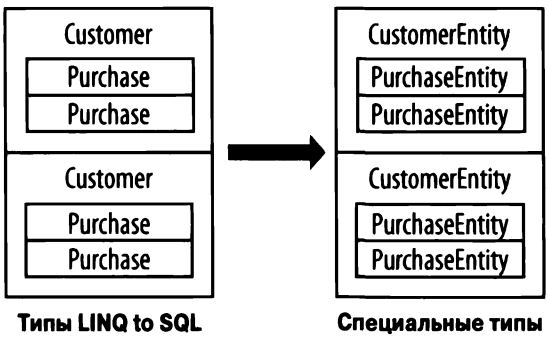



*Рис. 9.2. Проецирование иерархии объектов*

## SelectMany

| Аргумент | Тип |
|---|---|
| Исходная последовательность | `IEnumerable<TSource>` |
| Селектор результатов | `TSource => IEnumerable<TResult>` или `(TSource,int) => IEnumerable<TResult>`* |

* Запрещено в случае LINQ to SQL и Entity Framework.

### Синтаксис запросов

```
from идентификатор1 in перечислимое-выражение1
from идентификатор2 in перечислимое-выражение2
...
```

### Реализация в Enumerable

```
public static IEnumerable<TResult> SelectMany<TSource,TResult>
  (IEnumerable<TSource> source,
   Func <TSource,IEnumerable<TResult>> selector)
{
  foreach (TSource element in source)
    foreach (TResult subElement in selector (element))
      yield return subElement;
}
```

## Обзор

Операция SelectMany объединяет подпоследовательности в единую выходную последовательность.

Вспомните, что для каждого входного элемента операция Select выдает в точности один выходной элемент. Напротив, операция SelectMany выдает 0..*n* выходных элементов. Элементы 0..*n* берутся из подпоследовательности или дочерней последовательности, которую должно выпускать лямбда-выражение.

Операция SelectMany может использоваться для расширения дочерних последовательностей, выравнивания вложенных последовательностей и соединения двух коллекций в плоскую выходную последовательность. Применяя аналогию с конвейерными лентами, SelectMany помещает свежий материал на конвейерную ленту. Благодаря операции SelectMany каждый входной элемент является *спусковым механизмом* для подачи свежего материала. Свежий материал выпускается лямбда-выражением селектора и должен быть последовательностью. Другими словами, лямбда-выражение должно выдавать *дочернюю последовательность* для каждого входного *элемента*. Окончательным результатом будет объединение дочерних последовательностей, выпущенных для всех входных элементов.

Начнем с простого примера. Предположим, что имеется массив имен следующего вида:

```
string[] fullNames = { "Anne Williams", "John Fred Smith", "Sue Green" };
```

который необходимо преобразовать в одиночную плоскую коллекцию слов:

```
"Anne", "Williams", "John", "Fred", "Smith", "Sue", Green"
```

Для решения задачи идеально подходит операция SelectMany, т.к. мы сопоставляем каждый входной элемент с переменным количеством выходных элементов. Все, что потребуется сделать – построить выражение селектора, которое преобразует каждый входной элемент в дочернюю последовательность. Для этого используется метод string.Split, который берет строку и разбивает ее на слова, выпуская результат в виде массива:

```
string testInputElement = "Anne Williams";
string[] childSequence  = testInputElement.Split();

// childSequence содержит { "Anne", "Williams" };
```

Итак, ниже приведен запрос SelectMany и результат:

```
IEnumerable<string> query = fullNames.SelectMany (name => name.Split());

foreach (string name in query)
  Console.Write (name + "|");  // Anne|Williams|John|Fred|Smith|Sue|Green|
```

Если заменить SelectMany операцией Select, то будет получен тот же самый результат, но в иерархической форме. Следующий запрос выдает последовательность строковых *массивов*, которая для своего перечисления требует вложенных операторов foreach:

```
IEnumerable<string[]> query = fullNames.Select (name => name.Split());

foreach (string[] stringArray in query)
  foreach (string name in stringArray)
    Console.Write (name + "|");
```

Преимущество SelectMany в том, что выдается одиночная *плоская* результирующая последовательность.

Операция SelectMany поддерживается в синтаксисе запросов и вызывается с помощью *дополнительного генератора* – другими словами, дополнительной конструкции from в запросе. В синтаксисе запросов ключевое слово from играет две разных роли. В начале запроса оно вводит исходную переменную диапазона и входную последовательность. В *любом другом месте* запроса оно транслируется в операцию SelectMany. Ниже показан наш запрос, представленный в синтаксисе запросов:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()  // Транслируется в операцию SelectMany
  select name;
```

Обратите внимание, что дополнительный генератор вводит новую переменную диапазона – в данном случае name. Тем не менее, старая переменная диапазона остается в области видимости, и мы можем работать с ними обеими.

### Множество переменных диапазона

В предыдущем примере переменные name и fullName остаются в области видимости до тех пор, пока не завершится запрос или не будет достигнута конструкция into. Расширенная область видимости упомянутых переменных представляет собой сценарий, в котором синтаксис запросов выигрывает у текучего синтаксиса.

Чтобы проиллюстрировать сказанное, мы можем взять предыдущий пример и включить fullName в финальную проекцию:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()
  select name + " came from " + fullName;

Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
...
```

"За кулисами" компилятор должен предпринять ряд трюков, чтобы обеспечить доступ к обеим переменным. Хороший способ оценить ситуацию – попытаться написать тот же запрос в текучем синтаксисе. Это сложно! Задача еще более усложнится, если перед проецированием поместить конструкцию where или orderby:

```
from fullName in fullNames
from name in fullName.Split()
orderby fullName, name
select name + " came from " + fullName;
```

Проблема в том, что операция SelectMany выдает плоскую последовательность дочерних элементов – в данном случае плоскую коллекцию слов. Исходный "внешний" элемент, из которого она поступает (fullName), утерян. Решение заключается в том, чтобы "передавать" внешний элемент с каждым дочерним элементом во временном анонимном типе:

```
from fullName in fullNames
from x in fullName.Split().Select (name => new { name, fullName } )
orderby x.fullName, x.name
select x.name + " came from " + x.fullName;
```

Единственное изменение здесь заключается в том, что каждый дочерний элемент (name) помещается в оболочку анонимного типа, который также содержит его

fullName. Это похоже на то, как распознается конструкция let. Ниже показано финальное преобразование в текучий синтаксис:

```
IEnumerable<string> query = fullNames
  .SelectMany (fName => fName.Split()
                        .Select (name => new { name, fName } ))
  .OrderBy (x => x.fName)
  .ThenBy  (x => x.name)
  .Select  (x => x.name + " came from " + x.fName);
```

## Мышление в терминах синтаксиса запросов

Как мы только что продемонстрировали, существуют веские причины применять синтаксис запросов, когда нужно работать с несколькими переменными диапазона. В таких случаях полезно не только использовать этот синтаксис, но также и думать непосредственно в его терминах.

При написании дополнительных генераторов применяются два базовых шаблона. Первый из них — *расширение и выравнивание подпоследовательностей*. Для этого в дополнительном генераторе производится обращение к свойству или методу с существующей переменной диапазона. Мы поступали так в предыдущем примере:

```
from fullName in fullNames
from name in fullName.Split()
```

Здесь мы расширили перечисление фамилий до перечисления слов. Аналогичный запрос LINQ к базам данных производится, когда необходимо развернуть свойства дочерних ассоциаций. Следующий запрос выводит список всех заказчиков вместе с их покупками:

```
IEnumerable<string> query = from c in dataContext.Customers
                            from p in c.Purchases
                            select c.Name + " bought a " + p.Description;

Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
...
```

Здесь мы расширили каждого заказчика в подпоследовательность покупок.

Второй шаблон предусматривает выполнение *декартова произведения* или *перекрестного соединения*, при котором каждый элемент одной последовательности сопоставляется с каждым элементом другой последовательности. Чтобы сделать это, потребуется ввести генератор, выражение селектора которого возвращает последовательность, не связанную с какой-то переменной диапазона:

```
int[] numbers = { 1, 2, 3 };  string[] letters = { "a", "b" };

IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;

РЕЗУЛЬТАТ: { "1a", "1b", "2a", "2b", "3a", "3b" }
```

Такой стиль запроса является основой *соединений* в стиле SelectMany.

## Выполнение соединений с помощью SelectMany

Операцию SelectMany можно использовать для соединения двух последовательностей, просто отфильтровывая результаты векторного произведения.

Например, предположим, что необходимо сопоставить игроков друг с другом в игре. Мы можем начать следующим образом:

```
string[] players = { "Tom", "Jay", "Mary" };
IEnumerable<string> query = from name1 in players
                            from name2 in players
                            select name1 + " vs " + name2;
РЕЗУЛЬТАТ: { "Tom vs Tom", "Tom vs Jay", "Tom vs Mary",
             "Jay vs Tom", "Jay vs Jay", "Jay vs Mary",
             "Mary vs Tom", "Mary vs "Jay", "Mary vs Mary" }
```

Запрос читается так: "Для каждого игрока выполнить итерацию по каждому игроку, выбирая игрока 1 против игрока 2". Хотя мы получаем то, что запросили (перекрестное соединение), результаты бесполезны до тех пор, пока не будет добавлен фильтр:

```
IEnumerable<string> query = from name1 in players
                            from name2 in players
                            where name1.CompareTo (name2) < 0
                            orderby name1, name2
                            select name1 + " vs " + name2;
РЕЗУЛЬТАТ: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

Предикат фильтра образует *условие соединения*. Наш запрос можно назвать *неэквисоединением*, потому что в условии соединения операция эквивалентности не применяется.

Мы продемонстрируем оставшиеся типы соединений с помощью LINQ to SQL (они также будут работать с EF за исключением случаев, где явно используется поле внешнего ключа).

### SelectMany в LINQ to SQL и EF

Операция `SelectMany` в LINQ to SQL и EF может выполнять перекрестные соединения, неэквисоединения, внутренние соединения и левые внешние соединения. Как и `Select`, ее можно применять с предопределенными ассоциациями и произвольными отношениями. Отличие в том, что операция `SelectMany` возвращает плоский, а не иерархический результирующий набор.

Перекрестное соединение LINQ к базам данных записывается точно так же, как в предыдущем разделе. Следующий запрос сопоставляет каждого заказчика с каждой покупкой (перекрестное соединение):

```
var query = from c in dataContext.Customers
            from p in dataContext.Purchases
            select c.Name + " might have bought a " + p.Description;
```

Однако более типичной ситуацией является сопоставление заказчиков только с их собственными покупками. Это достигается добавлением конструкции `where` с предикатом соединения. В результате получается стандартное эквисоединение в стиле SQL:

```
var query = from c in dataContext.Customers
            from p in dataContext.Purchases
            where c.ID == p.CustomerID
            select c.Name + " bought a " + p.Description;
```

Такой запрос хорошо транслируется в SQL. В следующем разделе мы покажем, как его расширить для поддержки внешних соединений. Переписывание запросов подобного рода с использованием LINQ-операции `Join` в действительности делает их *менее* расширяемыми — в таком смысле язык LINQ противоположен SQL.

При наличии свойств ассоциаций для отношений в сущностях тот же самый запрос можно выразить путем развертывания подколлекции вместо фильтрации результатов векторного произведения:

```
from c in dataContext.Customers
from p in c.Purchases
select new { c.Name, p.Description };
```

Инфраструктура Entity Framework не открывает доступ к внешним ключам в сущностях, поэтому для распознанных отношений *должны* применяться их свойства ассоциаций, а не производиться соединение вручную, как мы поступали ранее.

Преимущество состоит в том, что мы устраняем предикат соединения. Мы перешли от фильтрации векторного произведения к развертыванию и выравниванию. Тем не менее, оба запроса дают в результате один и тот же код SQL.

Для дополнительной фильтрации к такому запросу можно добавлять конструкции `where`. Например, если нужны только заказчики, имена которых начинаются на "T", фильтрацию можно производить так:

```
from c in dataContext.Customers
where c.Name.StartsWith ("T")
from p in c.Purchases
select new { c.Name, p.Description };
```

Приведенный запрос LINQ к базам данных будет работать в равной степени хорошо, если переместить конструкцию `where` на одну строку ниже. Однако если это локальный запрос, то перемещение конструкции `where` вниз может сделать его менее эффективным. Для локальных запросов фильтрация должна выполняться *перед* соединением.

С помощью дополнительных конструкций `from` в полученную смесь можно вводить новые таблицы. Например, если каждая запись о покупке имеет дочерние строки деталей, то можно было бы построить плоский результирующий набор заказчиков с их покупками и деталями по каждой из них:

```
from c in dataContext.Customers
from p in c.Purchases
from pi in p.PurchaseItems
select new { c.Name, p.Description, pi.DetailLine };
```

Каждая конструкция `from` вводит новую *дочернюю* таблицу. Чтобы включить данные из *родительской* таблицы (через свойство ассоциации), не нужно добавлять конструкцию `from` – необходимо лишь перейти на это свойство. Скажем, если с каждым заказчиком связан продавец, имя которого требуется запросить, можно поступить следующим образом:

```
from c in dataContext.Customers
select new { Name = c.Name, SalesPerson = c.SalesPerson.Name };
```

В данном случае операция `SelectMany` не используется, т.к. отсутствуют подколлекции, подлежащие выравниванию. Родительское свойство ассоциации возвращает одиночный элемент.

### Выполнение внешних соединений с помощью SelectMany

Как было показано ранее, подзапрос Select выдает результат, аналогичный левому внешнему соединению:

```
from c in dataContext.Customers
select new {
            c.Name,
            Purchases = from p in c.Purchases
                        where p.Price > 1000
                        select new { p.Description, p.Price }
          };
```

В приведенном примере каждый внешний элемент (заказчик) включается независимо от того, совершались ли им какие-то покупки. Но предположим, что запрос переписан с применением операции SelectMany, а потому вместо иерархического результирующего набора можно получить одиночную плоскую коллекцию:

```
from c in dataContext.Customers
from p in c.Purchases
where p.Price > 1000
select new { c.Name, p.Description, p.Price };
```

В процессе выравнивания запроса мы перешли на внутреннее соединение: теперь включаются только такие заказчики, которые имеют одну или более дорогостоящих покупок. Чтобы получить левое внешнее соединение с плоским результирующим набором, мы должны применить к внутренней последовательности операцию запроса DefaultIfEmpty. Этот метод возвращает последовательность с единственным элементом null, когда входная последовательность не содержит элементов. Ниже показан такой запрос с опущенным предикатом цены:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new { c.Name, p.Description, Price = (decimal?) p.Price };
```

Данный запрос успешно работает с инфраструктурами LINQ to SQL и EF, возвращая всех заказчиков, даже если они вообще не совершали покупок. Но в случае его запуска как локального запроса произойдет аварийный отказ, потому что когда p равно null, обращения p.Description и p.Price генерируют исключение NullReferenceException. Мы можем сделать наш запрос надежным в обоих сценариях следующим образом:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

Теперь давайте займемся фильтром цены. Использовать конструкцию where, как мы поступали ранее, не получится, поскольку она выполняется *после* DefaultIfEmpty:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
where p.Price > 1000...
```

Корректное решение предусматривает сращивание конструкции Where *перед* DefaultIfEmpty с подзапросом:

```
from c in dataContext.Customers
from p in c.Purchases.Where (p => p.Price > 1000).DefaultIfEmpty()
select new {
              c.Name,
              Descript = p == null ? null : p.Description,
              Price = p == null ? (decimal?) null : p.Price
            };
```

Инфраструктуры LINQ to SQL и EF транслируют такой запрос в левое внешнее соединение. Это эффективный шаблон для написания запросов подобного рода.

Если вы привыкли к написанию внешних соединений на языке SQL, то можете не заметить более простой вариант с подзапросом `Select` для такого стиля запросов, а отдать предпочтение неудобному, но знакомому подходу, ориентированному на SQL, с плоским результирующим набором. Иерархический результирующий набор из подзапроса `Select` часто лучше подходит для запросов с внешними соединениями, потому что в таком случае отсутствуют дополнительные значения `null`, с которыми пришлось бы иметь дело.

# Выполнение соединения

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| `Join` | Применяет стратегию поиска для сопоставления элементов из двух коллекций, выпуская плоский результирующий набор | `INNER JOIN` |
| `GroupJoin` | Применяет стратегию поиска для сопоставления элементов из двух коллекций, выдавая иерархический результирующий набор | `INNER JOIN`, `LEFT OUTER JOIN` |
| `Zip` | Перечисляет две последовательности за раз (подобно застежке-молнии (zipper)), применяя функцию к каждой паре элементов | Генерируется исключение |

## `Join` и `GroupJoin`

**`IEnumerable<TOuter>`**, `IEnumerable<TInner>` → **`IEnumerable<TResult>`**

### Аргументы `Join`

| Аргумент | Тип |
|---|---|
| Внешняя последовательность | `IEnumerable<TOuter>` |
| Внутренняя последовательность | `IEnumerable<TInner>` |
| Внешний селектор ключей | `TOuter => TKey` |
| Внутренний селектор ключей | `TInner => TKey` |
| Селектор результатов | `(TOuter,TInner) => TResult` |

### Аргументы GroupJoin

| Аргумент | Тип |
|---|---|
| Внешняя последовательность | IEnumerable<TOuter> |
| Внутренняя последовательность | IEnumerable<TInner> |
| Внешний селектор ключей | TOuter => TKey |
| Внутренний селектор ключей | TInner => TKey |
| Селектор результатов | (TOuter, **IEnumerable<TInner>**) => TResult |

## Синтаксис запросов

```
from внешняя-переменная in внешнее-перечисление
join внутренняя-переменная in внутренее-перечисление
  on внешнее-выражение-ключей equals внутреннее-выражение-ключей
[ into идентификатор ]
```

## Обзор

Операции Join и GroupJoin объединяют две входные последовательности в единственную выходную последовательность. Операция Join выпускает плоский вывод, а GroupJoin – иерархический.

Операции Join и GroupJoin поддерживают стратегию, альтернативную операциям Select и SelectMany. Преимущество Join и GroupJoin связано с эффективным выполнением на локальных коллекциях в памяти, т.к. они сначала загружают внутреннюю последовательность в объект Lookup с ключами, устраняя необходимость в повторяющемся перечислении по всем внутренним элементам. Недостаток их в том, что они предлагают эквивалент только для внутренних и левых внешних соединений; перекрестные соединения и неэквисоединения по-прежнему должны делаться с помощью Select/SelectMany. В запросах LINQ to SQL и Entity Framework операции Join и GroupJoin не имеют реальных преимуществ перед Select и SelectMany.

В табл. 9.1 приведена сводка по отличиям между стратегиями соединения.

**Таблица 9.1. Стратегии соединения**

| Стратегия | Форма результатов | Эффективность локальных запросов | Внутренние соединения | Левые внешние соединения | Перекрестные соединения | Неэквисоединения |
|---|---|---|---|---|---|---|
| Select + SelectMany | Плоская | Плохая | Да | Да | Да | Да |
| Select + Select | Вложенная | Плохая | Да | Да | Да | Да |
| Join | Плоская | Хорошая | Да | – | – | – |
| GroupJoin | Вложенная | Хорошая | Да | Да | – | – |
| GroupJoin + SelectMany | Плоская | Хорошая | Да | Да | – | – |

### Join

Операция Join выполняет внутреннее соединение, выпуская плоскую выходную последовательность.

Инфраструктура Entity Framework скрывает поля внешних ключей, поэтому выполнить соединение по естественным отношениям вручную не получится (взамен можно производить запрос по свойствам ассоциаций, как было описано в предшествующих двух разделах).

Продемонстрировать работу Join проще всего в LINQ to SQL. Следующий запрос выводит список всех заказчиков вместе с их покупками без использования свойства ассоциации:

```
IQueryable<string> query =
  from c in dataContext.Customers
  join p in dataContext.Purchases on c.ID equals p.CustomerID
  select c.Name + " bought a " + p.Description;
```

Результаты совпадают с теми, которые были бы получены из запроса в стиле SelectMany:

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
```

Чтобы увидеть преимущество операции Join перед SelectMany, мы должны преобразовать запрос в локальный. В целях демонстрации мы сначала скопируем всех заказчиков и покупки в массивы, после чего выполним запросы к этим массивам:

```
Customer[] customers = dataContext.Customers.ToArray();
Purchase[] purchases = dataContext.Purchases.ToArray();
var slowQuery = from c in customers
                from p in purchases where c.ID == p.CustomerID
                select c.Name + " bought a " + p.Description;

var fastQuery = from c in customers
                join p in purchases on c.ID equals p.CustomerID
                select c.Name + " bought a " + p.Description;
```

Хотя оба запроса выдают одинаковые результаты, запрос Join заметно быстрее, потому что его реализация в классе Enumerable предварительно загружает внутреннюю коллекцию (purchases) в объект Lookup с ключами.

Синтаксис запросов для конструкции join может быть записан в общем случае так:

```
join внутренняя-переменная in внутренняя-последовательность
  on внешнее-выражение-ключей equals внутреннее-выражение-ключей
```

Операции соединений в LINQ проводят различие между *внутренней последовательностью* и *внешней последовательностью*. Вот что они означают с точки зрения синтаксиса.

- *Внешняя последовательность* — это входная последовательность (в данном случае customers).
- *Внутренняя последовательность* — это введенная вами новая коллекция (в данном случае purchases).

Операция Join выполняет внутренние соединения, а значит заказчики, не имеющие связанных с ними покупок, из вывода исключаются. При внутренних соединениях внутреннюю и внешнюю последовательности в запросе можно менять местами и по-прежнему получать те же самые результаты:

```
from p in purchases       // p теперь внешняя последовательность
join c in customers on p.CustomerID equals c.ID  // c теперь внутренняя
                                                 // последовательность
...
```

В запрос можно добавлять дополнительные конструкции join. Если, например, каждая запись о покупке имеет один или более элементов, тогда выполнить соединение элементов покупок можно следующим образом:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID        // первое соединение
join pi in purchaseItems on p.ID equals pi.PurchaseID  // второе соединение
...
```

Здесь purchases действует в качестве *внутренней* последовательности в первом соединении и в качестве *внешней* — во втором. Получить те же самые результаты (неэффективным способом) можно с применением вложенных операторов foreach:

```
foreach (Customer c in customers)
  foreach (Purchase p in purchases)
    if (c.ID == p.CustomerID)
      foreach (PurchaseItem pi in purchaseItems)
        if (p.ID == pi.PurchaseID)
          Console.WriteLine (c.Name + "," + p.Price + "," + pi.Detail);
```

В синтаксисе запросов переменные из более ранних соединений остаются в области видимости — точно так происходит и в запросах стиля SelectMany. Кроме того, между конструкциями join разрешено вставлять конструкции where и let.

### Выполнение соединений по нескольким ключам

Можно выполнять соединение по нескольким ключам с помощью анонимных типов:

```
from x in sequenceX
join y in sequenceY on new { K1 = x.Prop1, K2 = x.Prop2 }
                equals new { K1 = y.Prop3, K2 = y.Prop4 }
...
```

Чтобы прием работал, два анонимных типа должны быть идентично структурированными. Компилятор затем реализует каждый из них с помощью одного и того же внутреннего типа, делая соединяемые ключи совместимыми.

### Выполнение соединений в текучем синтаксисе

Показанное ниже соединение в синтаксисе запросов:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
select new { c.Name, p.Description, p.Price };
```

можно выразить с помощью текучего синтаксиса:

```
customers.Join (          // внешняя коллекция
     purchases,           // внутренняя коллекция
     c => c.ID,           // внешний селектор ключей
     p => p.CustomerID,   // внутренний селектор ключей
     (c, p) => new
       { c.Name, p.Description, p.Price }  // селектор результатов
);
```

Выражение селектора результатов в конце создает каждый элемент в выходной последовательности. Если перед проецированием присутствуют дополнительные конструкции, такие как orderby в данном примере:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
orderby p.Price
select c.Name + " bought a " + p.Description;
```

тогда для представления запроса в текучем синтаксисе потребуется создать временный анонимный тип внутри селектора результатов. Такой прием сохраняет c и p в области видимости, следуя соединению:

```
customers.Join (                    // внешняя коллекция
     purchases,                     // внутренняя коллекция
     c => c.ID,                     // внешний селектор ключей
     p => p.CustomerID,             // внутренний селектор ключей
     (c, p) => new { c, p } )       // селектор результатов
  .OrderBy (x => x.p.Price)
  .Select  (x => x.c.Name + " bought a " + x.p.Description);
```

При выполнении соединений синтаксис запросов обычно предпочтительнее; он требует менее кропотливой работы.

### GroupJoin

Операция GroupJoin делает то же самое, что и Join, но вместо плоского результата она выдает иерархический результат, сгруппированный по каждому внешнему элементу. Она также позволяет выполнять левые внешние соединения.

Синтаксис запросов для GroupJoin такой же, как и для Join, но за ним следует ключевое слово into.

Вот простейший пример:

```
IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  into custPurchases
  select custPurchases;  // custPurchases является последовательностью
```

Конструкция into транслируется в операцию GroupJoin, только когда она появляется непосредственно после конструкции join. После конструкции select или group она означает *продолжение запроса*. Указанные два сценария использования ключевого слова into значительно отличаются, хотя и обладают одной общей характеристикой: в обоих случаях вводится новая переменная диапазона.

Результатом будет последовательность последовательностей, для которой можно выполнить перечисление:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

Тем не менее, это не особенно полезно, т.к. purchaseSequence не имеет ссылок на заказчика. Чаще всего следует поступать так:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
```

```
into custPurchases
select new { CustName = c.Name, custPurchases };
```

Результаты будут такими же, как у приведенного ниже (неэффективного) подзапроса Select:

```
from c in customers
select new
{
  CustName = c.Name,
  custPurchases = purchases.Where (p => c.ID == p.CustomerID)
};
```

По умолчанию операция GroupJoin является эквивалентом левого внешнего соединения. Чтобы получить внутреннее соединение, где заказчики без покупок исключены, понадобится реализовать фильтрацию по custPurchases:

```
from c in customers join p in purchases on c.ID equals p.CustomerID
into custPurchases
where custPurchases.Any()
select ...
```

Конструкции после into в GroupJoin оперируют на *подпоследовательностях* внутренних дочерних элементов, а не на *индивидуальных* дочерних элементах. Это значит, что для фильтрации отдельных покупок потребуется вызвать операцию Where *перед* соединением:

```
from c in customers
join p in purchases.Where (p2 => p2.Price > 1000)
  on c.ID equals p.CustomerID
into custPurchases ...
```

С помощью операции GroupJoin можно конструировать лямбда-выражения, как это делается посредством Join.

### Плоские внешние соединения

Когда требуется и внешнее соединение, и плоский результирующий набор, возникает дилемма. Операция GroupJoin обеспечивает внешнее соединение, а Join дает плоский результирующий набор. Решение заключается в том, чтобы сначала вызвать GroupJoin, затем метод DefaultIfEmpty на каждой дочерней последовательности и, наконец, метод SelectMany на результате:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID into custPurchases
from cp in custPurchases.DefaultIfEmpty()
select new
{
  CustName = c.Name,
  Price = cp == null ? (decimal?) null : cp.Price
};
```

Метод DefaultIfEmpty возвращает последовательность с единственным значением null, если подпоследовательность покупок пуста. Вторая конструкция from транслируется в вызов метода SelectMany. В такой роли она *развертывает и выравнивает* все подпоследовательности покупок, объединяя их в единую последовательность *элементов* покупок.

### Выполнение соединений с помощью объектов Lookup

Методы Join и GroupJoin в Enumerable работают в два этапа. Во-первых, они загружают внутреннюю последовательность в объект Lookup. Во-вторых, они запрашивают внешнюю последовательность в комбинации с объектом Lookup.

*Объект Lookup* — это последовательность групп, в которую можно получать доступ напрямую по ключу. По-другому его можно воспринимать как словарь последовательностей — словарь, который может принимать множество элементов для каждого ключа (иногда его называют *мультисловарем*). Объекты Lookup предназначены только для чтения и определены в соответствии со следующим интерфейсом:

```
public interface ILookup<TKey,TElement> :
   IEnumerable<IGrouping<TKey,TElement>>, IEnumerable
{
  int Count { get; }
  bool Contains (TKey key);
  IEnumerable<TElement> this [TKey key] { get; }
}
```

Операции соединений (как и все остальные операции, выдающие последовательности) поддерживают семантику отложенного или ленивого выполнения. Это значит, что объект Lookup не будет построен до тех пор, пока вы не начнете перечисление выходной последовательности (и тогда сразу же строится *целый* объект Lookup).

Имея дело с локальными коллекциями, в качестве альтернативы применению операций соединения объекты Lookup можно создавать и запрашивать вручную. Такой подход обладает парой преимуществ:

- один и тот же объект Lookup можно многократно использовать во множестве запросов — равно как и в обычном императивном коде;
- выдача запросов к объекту Lookup — отличный способ понять, каким образом работают операции Join и GroupJoin.

Объект Lookup создается с помощью расширяющего метода ToLookup. Следующий код загружает все покупки в объект Lookup — с ключами в виде их идентификаторов CustomerID:

```
ILookup<int?,Purchase> purchLookup =
  purchases.ToLookup (p => p.CustomerID, p => p);
```

Первый аргумент выбирает ключ, а второй — объекты, которые должны загружаться в качестве значений в Lookup.

Чтение объекта Lookup довольно похоже на чтение словаря за исключением того, что индексатор возвращает *последовательность* соответствующих элементов, а не *одиночный* элемент. Приведенный ниже код перечисляет все покупки, сделанные заказчиком с идентификатором 1:

```
foreach (Purchase p in purchLookup [1])
  Console.WriteLine (p.Description);
```

При наличии объекта Lookup можно написать запросы SelectMany/Select, которые выполняются так же эффективно, как запросы Join/GroupJoin. Операция Join эквивалентна применению метода SelectMany на объекте Lookup:

```
from c in customers
from p in purchLookup [c.ID]
select new { c.Name, p.Description, p.Price };
```

```
Tom Bike 500
Tom Holiday 2000
Dick Bike 600
Dick Phone 300
...
```

Добавление вызова метода `DefaultIfEmpty` делает этот запрос внутренним соединением:

```
from c in customers
from p in purchLookup [c.ID].DefaultIfEmpty()
 select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
            };
```

Использование `GroupJoin` эквивалентно чтению объекта `Lookup` внутри проекции:

```
from c in customers
select new {
            CustName = c.Name,
            CustPurchases = purchLookup [c.ID]
           };
```

### Реализация в Enumerable

Ниже представлена простейшая допустимая реализация `Enumerable.Join`, не включающая проверку на предмет равенства `null`:

```
public static IEnumerable <TResult> Join
                              <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>   outer,
  IEnumerable <TInner>        inner,
  Func <TOuter,TKey>          outerKeySelector,
  Func <TInner,TKey>          innerKeySelector,
  Func <TOuter,TInner,TResult> resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    from innerItem in lookup [outerKeySelector (outerItem)]
    select resultSelector (outerItem, innerItem);
}
```

Реализация `GroupJoin` похожа на реализацию `Join`, но только проще:

```
public static IEnumerable <TResult> GroupJoin
                              <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>   outer,
  IEnumerable <TInner>        inner,
  Func <TOuter,TKey>          outerKeySelector,
  Func <TInner,TKey>          innerKeySelector,
  Func <TOuter,IEnumerable<TInner>,TResult>  resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    select resultSelector
     (outerItem, lookup [outerKeySelector (outerItem)]);
}
```

## Операция Zip

**IEnumerable<TFirst>, IEnumerable<TSecond> → IEnumerable<TResult>**

Операция Zip появилась в версии .NET Framework 4.0. Она выполняет перечисление двух последовательностей за раз (подобно застежке-молнии (zipper)) и возвращает последовательность, основанную на применении некоторой функции к каждой паре элементов. Например, следующий код:

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip (words, (n, w) => n + "=" + w);
```

выдает последовательность с такими элементами:

```
3=three
5=five
7=seven
```

Избыточные элементы в любой из входных последовательностей игнорируются. Операция Zip не поддерживается инфраструктурами EF и L2S.

# Упорядочение

**IEnumerable<TSource> → IOrderedEnumerable<TSource>**

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| OrderBy, ThenBy | Сортируют последовательность в возрастающем порядке | ORDER BY ... |
| OrderByDescending, ThenByDescending | Сортируют последовательность в убывающем порядке | ORDER BY ... DESC |
| Reverse | Возвращает последовательность в обратном порядке | Генерируется исключение |

Операции упорядочения возвращают те же самые элементы, но в другом порядке.

## OrderBy, OrderByDescending, ThenBy и ThenByDescending

### Аргументы OrderBy и OrderByDescending

| Аргумент | Тип |
|---|---|
| Входная последовательность | IEnumerable<TSource> |
| Селектор ключей | TSource => TKey |

Возвращаемый тип: IOrderedEnumerable<TSource>

### Аргументы ThenBy и ThenByDescending

| Аргумент | Тип |
|---|---|
| Входная последовательность | IOrderedEnumerable<TSource> |
| Селектор ключей | TSource => TKey |

### Синтаксис запросов

```
orderby выражение1 [descending] [, выражение2 [descending] ... ]
```

### Обзор

Операция `OrderBy` возвращает отсортированную версию входной последовательности, используя выражение `keySelector` для выполнения сравнений. Следующий запрос выдает последовательность имен в алфавитном порядке:

```
IEnumerable<string> query = names.OrderBy (s => s);
```

А здесь имена сортируются по их длине:

```
IEnumerable<string> query = names.OrderBy (s => s.Length);
// Результат: { "Jay", "Tom", "Mary", "Dick", "Harry" };
```

Относительный порядок элементов с одинаковыми ключами сортировки (в данном случае Jay/Tom и Mary/Dick) не определен, если только не добавить операцию `ThenBy`:

```
IEnumerable<string> query = names.OrderBy (s => s.Length).ThenBy (s => s);
// Результат: { "Jay", "Tom", "Dick", "Mary", "Harry" };
```

Операция `ThenBy` переупорядочивает только элементы, которые имеют один и тот же ключ сортировки из предшествующей сортировки. Допускается выстраивать в цепочку любое количество операций `ThenBy`. Следующий запрос сортирует сначала по длине, затем по второму символу и, наконец, по первому символу:

```
names.OrderBy (s => s.Length).ThenBy (s => s[1]).ThenBy (s => s[0]);
```

Ниже показан эквивалент в синтаксисе запросов:

```
from s in names
orderby s.Length, s[1], s[0]
select s;
```

Приведенная далее вариация *некорректна* – в действительности она будет упорядочивать сначала по `s[1]` и затем по `s.Length` (в случае запроса к базе данных она будет упорядочивать *только* по `s[1]` и отбрасывать первое упорядочение):

```
from s in names
orderby s.Length
orderby s[1]
...
```

Язык LINQ также предлагает операции `OrderByDescending` и `ThenByDescending`, которые делают то же самое, выдавая результаты в обратном порядке. Следующий запрос LINQ к базе данных извлекает покупки в убывающем порядке цен, выстраивая покупки с одинаковой ценой в алфавитном порядке:

```
dataContext.Purchases.OrderByDescending (p => p.Price)
                     .ThenBy (p => p.Description);
```

А вот он в синтаксисе запросов:

```
from p in dataContext.Purchases
orderby p.Price descending, p.Description
select p;
```

### Компараторы и сопоставления

В локальном запросе объекты селекторов ключей самостоятельно определяют алгоритм упорядочения через свою стандартную реализацию интерфейса IComparable (см. главу 7). Переопределить алгоритм сортировки можно путем передачи объекта реализации IComparer. Показанный ниже запрос выполняет сортировку, нечувствительную к регистру:

```
names.OrderBy (n => n, StringComparer.CurrentCultureIgnoreCase);
```

Передача компаратора не поддерживается в синтаксисе запросов, а также невозможна ни в LINQ to SQL, ни в EF. При запросе базы данных алгоритм сравнения определяется сопоставлением участвующего столбца. Если сопоставление чувствительно к регистру, то нечувствительную к регистру сортировку можно затребовать вызовом метода ToUpper в селекторе ключей:

```
from p in dataContext.Purchases
orderby p.Description.ToUpper()
select p;
```

### IOrderedEnumerable и IOrderedQueryable

Операции упорядочения возвращают специальные подтипы IEnumerable<T>. Операции упорядочения в классе Enumerable возвращают тип IOrderedEnumerable<TSource>, а операции упорядочения в классе Queryable — тип IOrderedQueryable<TSource>. Упомянутые подтипы позволяют с помощью последующей операции ThenBy уточнять, а не заменять существующее упорядочение.

Дополнительные члены, которые эти подтипы определяют, не открыты публично, так что они представляются как обычные последовательности. Тот факт, что они являются разными типами, вступает в игру при постепенном построении запросов:

```
IOrderedEnumerable<string> query1 = names.OrderBy (s => s.Length);
IOrderedEnumerable<string> query2 = query1.ThenBy (s => s);
```

Если взамен объявить переменную query1 как имеющую тип IEnumerable<string>, тогда вторая строка не скомпилируется — операция ThenBy требует на входе тип IOrderedEnumerable<string>. Чтобы не переживать по такому поводу, переменные диапазона можно типизировать неявно:

```
var query1 = names.OrderBy (s => s.Length);
var query2 = query1.ThenBy (s => s);
```

Однако неявная типизация может и сама создать проблемы. Следующий код не скомпилируется:

```
var query = names.OrderBy (s => s.Length);
query = query.Where (n => n.Length > 3);  // Ошибка на этапе компиляции
```

В качестве типа переменной query компилятор выводит IOrderedEnumerable <string>, основываясь на типе выходной последовательности операции OrderBy. Тем не менее, операция Where в следующей строке возвращает обычную реализацию IEnumerable<string>, которая не может быть присвоена query. Обойти проблему можно либо за счет явной типизации, либо путем вызова метода AsEnumerable после OrderBy:

```
var query = names.OrderBy (s => s.Length).AsEnumerable();
query = query.Where (n => n.Length > 3);  // Компилируется
```

Эквивалентом AsEnumerable в интерпретируемых запросах является метод AsQueryable.

# Группирование

**IEnumerable<TSource> → IEnumerable<IGrouping<TKey,TElement>>**

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| GroupBy | Группирует последовательность в подпоследовательности | GROUP BY |

## GroupBy

| Аргумент | Тип |
|---|---|
| Входная последовательность | IEnumerable<TSource> |
| Селектор ключей | TSource => TKey |
| Селектор элементов (необязательный) | TSource => TElement |
| Компаратор (необязательный) | IEqualityComparer<TKey> |

### Синтаксис запросов

group *выражение-элементов* by *выражение-ключей*

### Обзор

Операция GroupBy организует плоскую входную последовательность в последовательность *групп*. Например, приведенный ниже код организует все файлы в каталоге *c:\temp* по их расширениям:

```
string[] files = Directory.GetFiles ("c:\\temp");

IEnumerable<IGrouping<string,string>> query =
  files.GroupBy (file => Path.GetExtension (file));
```

Если неявная типизация удобнее, то можно записать так:

```
var query = files.GroupBy (file => Path.GetExtension (file));
```

А вот как выполнить перечисление результата:

```
foreach (IGrouping<string,string> grouping in query)
{
  Console.WriteLine ("Extension: " + grouping.Key);
  foreach (string filename in grouping)
    Console.WriteLine ("  -- " + filename);
}

Extension: .pdf
  -- chapter03.pdf
  -- chapter04.pdf
Extension: .doc
  -- todo.doc
  -- menu.doc
  -- Copy of menu.doc
...
```

Метод Enumerable.GroupBy работает путем чтения входных элементов во временный словарь списков, так что все элементы с одинаковыми ключами попадают в один и тот же подсписок. Затем выдается последовательность *групп*. Группа – это последовательность со свойством Key:

```
public interface IGrouping <TKey,TElement> : IEnumerable<TElement>,
                                             IEnumerable
{
  TKey Key { get; }   // Ключ применяется к подпоследовательности
                      // как к единому целому
}
```

По умолчанию элементы в каждой группе являются нетрансформированными входными элементами, если только не указан аргумент elementSelector. Следующий код проецирует каждый входной элемент в верхний регистр:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper());
```

Аргумент elementSelector не зависит от keySelector. В данном случае это означает, что ключ каждой группы по-прежнему представлен в первоначальном регистре:

```
Extension: .pdf
  -- CHAPTER03.PDF
  -- CHAPTER04.PDF
Extension: .doc
  -- TODO.DOC
```

Обратите внимание, что подколлекции не выдаются в алфавитном порядке ключей. Операция GroupBy только группирует, не выполняя *сортировку*; на самом деле она предохраняет исходное упорядочение. Чтобы отсортировать, потребуется добавить операцию OrderBy:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper())
     .OrderBy (grouping => grouping.Key);
```

Операция GroupBy имеет простую и прямую трансляцию в синтаксис запросов:

```
group выражение-элементов by выражение-ключей
```

Ниже приведен наш пример, представленный в синтаксисе запросов:

```
from file in files
group file.ToUpper() by Path.GetExtension (file);
```

Как и select, конструкция group "заканчивает" запрос – если только не была добавлена конструкция продолжения запроса:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
orderby grouping.Key
select grouping;
```

Продолжения запросов часто удобны в запросах group by. Следующий запрос отфильтровывает группы, которые содержат менее пяти файлов:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
where grouping.Count() >= 5
select grouping;
```

Конструкция where после group by является эквивалентом конструкции HAVING в SQL. Она применяется к каждой подпоследовательности или группе как к единому целому, а не к ее индивидуальным элементам.

Иногда интересует только результат агрегирования на группах, потому подпоследовательности можно отбросить:

```
string[] votes = { "Bush", "Gore", "Gore", "Bush", "Bush" };
IEnumerable<string> query = from vote in votes
                            group vote by vote into g
                            orderby g.Count() descending
                            select g.Key;
string winner = query.First();  // Bush
```

### GroupBy в LINQ to SQL и EF

При запрашивании базы данных группирование работает аналогичным образом. Однако если есть настроенные свойства ассоциаций, то вы обнаружите, что потребность в группировании возникает менее часто, чем при работе со стандартным языком SQL. Например, для выборки заказчиков с не менее чем двумя покупками группирование не понадобится; запрос может быть записан так:

```
from c in dataContext.Customers
where c.Purchases.Count >= 2
select c.Name + " has made " + c.Purchases.Count + " purchases";
```

Примером, когда может использоваться группирование, служит вывод списка итоговых продаж по годам:

```
from p in dataContext.Purchases
group p.Price by p.Date.Year into salesByYear
select new {
             Year       = salesByYear.Key,
             TotalValue = salesByYear.Sum()
           };
```

Группирование в LINQ отличается большей мощностью, чем конструкция GROUP BY в SQL. Например, вполне законно извлечь все строки с деталями покупок безо всякого агрегирования:

```
from p in dataContext.Purchases
group p by p.Date.Year
```

Такой прием хорошо работает в EF, но в L2S приводит к избыточному обращению к серверу. Проблему легко обойти за счет вызова метода AsEnumerable прямо перед группированием, в результате чего группирование произойдет на стороне клиента. Это будет не менее эффективным до тех пор, пока любая фильтрация выполняется *перед* группированием, так что из сервера извлекаются только те данные, которые необходимы.

Еще одно отклонение от традиционного языка SQL связано с отсутствием обязательного проецирования переменных или выражений, участвующих в группировании или сортировке.

### Группирование по нескольким ключам

Можно группировать по составному ключу с применением анонимного типа:

```
from n in names
group n by new { FirstLetter = n[0], Length = n.Length };
```

### Специальные компараторы эквивалентности

В локальном запросе для изменения алгоритма сравнения ключей методу `GroupBy` можно передавать специальный компаратор эквивалентности. Однако такое действие требуется редко, потому что модификации выражения селектора ключей обычно вполне достаточно. Например, следующий запрос создает группирование, нечувствительное к регистру:

```
group name by name.ToUpper()
```

# Операции над множествами

**`IEnumerable<TSource>`, `IEnumerable<TSource>` → `IEnumerable<TSource>`**

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| `Concat` | Возвращает результат конкатенации элементов в каждой из двух последовательностей | `UNION ALL` |
| `Union` | Возвращает результат конкатенации элементов в каждой из двух последовательностей, исключая дубликаты | `UNION` |
| `Intersect` | Возвращает элементы, присутствующие в обеих последовательностях | `WHERE ... IN (...)` |
| `Except` | Возвращает элементы, присутствующие в первой, но не во второй последовательности | `EXCEPT` или `WHERE ... NOT IN (...)` |

## `Concat` и `Union`

Операция `Concat` возвращает все элементы из первой последовательности, за которыми следуют все элементы из второй последовательности. Операция `Union` делает то же самое, но удаляет любые дубликаты:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
  concat = seq1.Concat (seq2),      // { 1, 2, 3, 3, 4, 5 }
  union  = seq1.Union  (seq2);      // { 1, 2, 3, 4, 5 }
```

Явное указание аргумента типа удобно, когда последовательности типизированы по-разному, но элементы имеют общий базовый тип. Например, благодаря API-интерфейсу рефлексии (глава 19) методы и свойства представлены с помощью классов `MethodInfo` и `PropertyInfo`, которые имеют общий базовый класс по имени `MemberInfo`. Мы можем выполнить конкатенацию методов и свойств, явно указывая базовый класс при вызове `Concat`:

```
MethodInfo[] methods = typeof (string).GetMethods();
PropertyInfo[] props = typeof (string).GetProperties();
IEnumerable<MemberInfo> both = methods.Concat<MemberInfo> (props);
```

В следующем примере мы фильтруем методы перед конкатенацией:

```
var methods = typeof (string).GetMethods().Where (m => !m.IsSpecialName);
var props = typeof (string).GetProperties();
var both = methods.Concat<MemberInfo> (props);
```

Данный пример полагается на вариантность параметров типа в интерфейсе: `methods` относится к типу `IEnumerable<MethodInfo>`, который требует ковариант-

ного преобразования в тип IEnumerable<MemberInfo>. Пример является хорошей иллюстрацией того, как вариантность делает поведение типов более ожидаемым.

### Intersect и Except

Операция Intersect возвращает элементы, имеющиеся в двух последовательностях. Операция Except возвращает элементы из первой последовательности, которые *не* присутствуют во второй последовательности:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
  commonality = seq1.Intersect (seq2),  //  { 3 }
  difference1 = seq1.Except    (seq2),  //  { 1, 2 }
  difference2 = seq2.Except    (seq1);  //  { 4, 5 }
```

Внутренне метод Enumerable.Except загружает все элементы первой последовательности в словарь, после чего удаляет из словаря элементы, присутствующие во второй последовательности. Эквивалентом такой операции в SQL является подзапрос NOT EXISTS или NOT IN:

```
SELECT number FROM numbers1Table
WHERE number NOT IN (SELECT number FROM numbers2Table)
```

## Методы преобразования

Язык LINQ в основном имеет дело с последовательностями — другими словами, с коллекциями типа IEnumerable<T>. Методы преобразования преобразуют коллекции IEnumerable<T> в коллекции других типов и наоборот.

| Метод | Описание |
|---|---|
| OfType | Преобразует IEnumerable в IEnumerable<T>, отбрасывая некорректно типизированные элементы |
| Cast | Преобразует IEnumerable в IEnumerable<T>, генерируя исключение при наличии некорректно типизированных элементов |
| ToArray | Преобразует IEnumerable<T> в T[] |
| ToList | Преобразует IEnumerable<T> в List<T> |
| ToDictionary | Преобразует IEnumerable<T> в Dictionary<TKey,TValue> |
| ToLookup | Преобразует IEnumerable<T> в ILookup<TKey,TElement> |
| AsEnumerable | Приводит вниз к IEnumerable<T> |
| AsQueryable | Приводит или преобразует в IQueryable<T> |

### OfType и Cast

Методы OfType и Cast принимают необобщенную коллекцию IEnumerable и выдают обобщенную последовательность IEnumerable<T>, которую впоследствии можно запрашивать:

```
ArrayList classicList = new ArrayList();  // в System.Collections
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

Когда методы Cast и OfType встречают входной элемент, который имеет несовместимый тип, их поведение отличается. Метод Cast генерирует исключение, а OfType игнорирует такой элемент. Продолжим предыдущий пример:

```
DateTime offender = DateTime.Now;
classicList.Add (offender);
IEnumerable<int>
  sequence2 = classicList.OfType<int>(),     // Нормально - проблемный
                                             // элемент DateTime игнорируется
  sequence3 = classicList.Cast<int>();       // Генерируется исключение
```

Правила совместимости элементов точно соответствуют правилам для операции is в языке C# и, следовательно, предусматривают только ссылочные и распаковывающие преобразования. Мы можем увидеть это, взглянув на внутреннюю реализацию OfType:

```
public static IEnumerable<TSource> OfType <TSource> (IEnumerable source)
{
  foreach (object element in source)
    if (element is TSource)
      yield return (TSource)element;
}
```

Метод Cast имеет идентичную реализацию за исключением того, что опускает проверку на предмет совместимости типов:

```
public static IEnumerable<TSource> Cast <TSource> (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

Из реализаций следует, что использовать Cast для выполнения числовых или специальных преобразований нельзя (взамен для них придется выполнять операцию Select). Другими словами, операция Cast не настолько гибкая, как операция приведения C#:

```
int i = 3;
long l = i;          // Неявное числовое преобразование int в long
int i2 = (int) l;    // Явное числовое преобразование long в int
```

Продемонстрировать сказанное можно, попробовав применить операции OfType или Cast для преобразования последовательности значений int в последовательность значений long:

```
int[] integers = { 1, 2, 3 };

IEnumerable<long> test1 = integers.OfType<long>();
IEnumerable<long> test2 = integers.Cast<long>();
```

При перечислении test1 выдает ноль элементов, а test2 генерирует исключение. Просмотр реализации метода OfType четко проясняет причину. После подстановки TSource мы получаем следующее выражение:

```
(element is long)
```

которое возвращает false, когда element имеет тип int, из-за отсутствия отношения наследования.

Причина того, что test2 генерирует исключение при перечислении, несколько тоньше. В реализации метода Cast обратите внимание на то, что element имеет тип object. Когда TSource является типом значения, среда CLR предполагает, что это *распаковывающее преобразование*, и синтезирует метод, который воспроизводит сценарий, описанный в разделе "Упаковка и распаковка" главы 3:

```
int value = 123;
object element = value;
long result = (long) element;  // Генерируется исключение
```

Поскольку переменная element объявлена с типом object, выполняется приведение object к long (распаковка), а не числовое преобразование int в long. Распаковывающие операции требуют точного соответствия типов, так что распаковка object в long терпит неудачу, когда элемент является int.

Как предполагалось ранее, решение заключается в использовании обычной операции Select:

```
IEnumerable<long> castLong = integers.Select (s => (long) s);
```

Операции OfType и Cast также удобны для приведения вниз элементов в обобщенной входной коллекции. Например, если имеется входная коллекция типа IEnumerable<Fruit> (фрукты), то OfType<Apple> (яблоки) возвратит только яблоки. Это особенно полезно в LINQ to XML (глава 10).

Операция Cast поддерживается в синтаксисе запросов: понадобится лишь предварить переменную диапазона нужным типом:

```
from TreeNode node in myTreeView.Nodes
...
```

## ToArray, ToList, ToDictionary и ToLookup

Операции ToArray и ToList выдают результаты в массив или обобщенный список. Они вызывают немедленное перечисление входной последовательности (если только не применяются косвенно через подзапрос или дерево выражения). За примерами обращайтесь в раздел "Отложенное выполнение" главы 8.

Операции ToDictionary и ToLookup принимают описанные ниже аргументы.

| Аргумент | Тип |
|---|---|
| Входная последовательность | IEnumerable<TSource> |
| Селектор ключей | TSource => TKey |
| Селектор элементов (необязательный) | TSource => TElement |
| Компаратор (необязательный) | IEqualityComparer<TKey> |

Операция ToDictionary также приводит к немедленному перечислению последовательности с записью результатов в обобщенный словарь Dictionary. Предоставляемое выражение селектора ключей (keySelector) должно вычисляться как уникальное значение для каждого элемента во входной последовательности; в противном случае генерируется исключение. Напротив, операция ToLookup позволяет множеству элементов иметь один и тот же ключ. Объекты Lookup рассматривались в разделе "Выполнение соединений с помощью объектов Lookup" ранее в главе.

### AsEnumerable и AsQueryable

Операция AsEnumerable выполняет приведение вверх последовательности к типу IEnumerable<T>, заставляя компилятор привязывать последующие операции запросов к методам из класса Enumerable, а не Queryable. За примером обращайтесь в раздел "Комбинирование интерпретируемых и локальных запросов" главы 8.

Операция AsQueryable выполняет приведение вниз последовательности к типу IQueryable<T>, если последовательность реализует этот интерфейс. В противном случае операция создает оболочку IQueryable<T> вокруг локального запроса.

## Операции над элементами

**IEnumerable<TSource> → TSource**

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| First, FirstOrDefault | Возвращают первый элемент в последовательности, необязательно удовлетворяющий предикату | SELECT TOP 1... ORDER BY... |
| Last, LastOrDefault | Возвращают последний элемент в последовательности, необязательно удовлетворяющий предикату | SELECT TOP 1... ORDER BY...DESC |
| Single, SingleOrDefault | Эквивалентны операциям First/FirstOrDefault, но генерируют исключение, если обнаружено более одного совпадения | |
| ElementAt, ElementAtOrDefault | Возвращают элемент в указанной позиции | Генерируется исключение |
| DefaultIfEmpty | Возвращает одноэлементную последовательность, значением которой является default(TSource), если последовательность не содержит элементов | OUTER JOIN |

Методы с именами, завершающимися на OrDefault, возвращают default(TSource), а не генерируют исключение, если входная последовательность является пустой или если нет элементов, соответствующих заданному предикату.

Значение default(TSource) равно null для элементов ссылочных типов, false — для элементов типа bool и 0 — для элементов числовых типов.

### First, Last и Single

| Аргумент | Тип |
|---|---|
| Входная последовательность | IEnumerable<TSource> |
| Предикат (необязательный) | TSource => bool |

В следующем примере демонстрируется работа операций First и Last:

```
int[] numbers  = { 1, 2, 3, 4, 5 };

int first      = numbers.First();              // 1
int last       = numbers.Last();               // 5
```

```
int firstEven  = numbers.First  (n => n % 2 == 0);     // 2
int lastEven   = numbers.Last   (n => n % 2 == 0);     // 4
```

Ниже проиллюстрирована работа операций `First` и `FirstOrDefault`:

```
int firstBigError  = numbers.First  (n => n > 10); // Генерируется исключение
int firstBigNumber = numbers.FirstOrDefault (n => n > 10);  // 0
```

Чтобы не генерировалось исключение, операция `Single` требует наличия точно одного совпадающего элемента, а `SingleOrDefault` — одного *или нуля* совпадающих элементов:

```
int onlyDivBy3 = numbers.Single (n => n % 3 == 0);     // 3
int divBy2Err  = numbers.Single (n => n % 2 == 0);     // Ошибка: совпадение
                                                       // дают 2 и 4

int singleError = numbers.Single          (n => n > 10);        // Ошибка
int noMatches   = numbers.SingleOrDefault (n => n > 10);        // 0
int divBy2Error = numbers.SingleOrDefault (n => n % 2 == 0);    // Ошибка
```

В данном семействе операций над элементами `Single` является наиболее "придирчивой". С другой стороны, операции `FirstOrDefault` и `LastOrDefault` наиболее толерантны.

В LINQ to SQL и EF операция `Single` часто используется для извлечения строки из таблицы по первичному ключу:

```
Customer cust = dataContext.Customers.Single (c => c.ID == 3);
```

## ElementAt

| Аргумент | Тип |
|---|---|
| Входная последовательность | `IEnumerable<TSource>` |
| Индекс элемента для возврата | `int` |

Операция `ElementAt` извлекает *n*-ный элемент из последовательности:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int third      = numbers.ElementAt (2);          // 3
int tenthError = numbers.ElementAt (9);          // Генерируется исключение
int tenth      = numbers.ElementAtOrDefault (9); // 0
```

Метод `Enumerable.ElementAt` написан так, что если входная последовательность реализует интерфейс `IList<T>`, тогда он вызывает индексатор, определенный в `IList<T>`. В противном случае он выполняет перечисление *n* раз и затем возвращает следующий элемент. Инфраструктуры LINQ to SQL и EF не поддерживают операцию `ElementAt`.

## DefaultIfEmpty

Операция `DefaultIfEmpty` возвращает последовательность с единственным элементом, значением которого будет `default(TSource)`, если входная последовательность не содержит элементов. В противном случае она возвращает неизмененную входную последовательность. Операция `DefaultIfEmpty` применяется при написании плоских внутренних соединений: за деталями обращайтесь в разделы "Выполнение внешних соединений с помощью `SelectMany`" и "Плоские внешние соединения" ранее в главе.

# Методы агрегирования

**IEnumerable<TSource>** → *скаляр*

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| `Count`, `LongCount` | Возвращают количество элементов во входной последовательности, необязательно удовлетворяющих предикату | `COUNT (...)` |
| `Min`, `Max` | Возвращают наименьший или наибольший элемент в последовательности | `MIN (...)`, `MAX (...)` |
| `Sum`, `Average` | Подсчитывают числовую сумму или среднее значение для элементов в последовательности | `SUM (...)`, `AVG (...)` |
| `Aggregate` | Выполняет специальное агрегирование | Генерируется исключение |

## Count и LongCount

| Аргумент | Тип |
|---|---|
| Входная последовательность | `IEnumerable<TSource>` |
| Предикат (необязательный) | `TSource => bool` |

Операция `Count` просто выполняет перечисление последовательности, возвращая количество элементов:

```
int fullCount = new int[] { 5, 6, 7 }.Count();  // 3
```

Внутренняя реализация метода `Enumerable.Count` проверяет входную последовательность на предмет реализации ею интерфейса `ICollection<T>`. Если он реализован, тогда просто производится обращение к свойству `ICollection<T>.Count`. В противном случае осуществляется перечисление по элементам последовательности с инкрементированием счетчика.

Можно дополнительно указать предикат:

```
int digitCount = "pa55w0rd".Count (c => char.IsDigit (c));  // 3
```

Операция `LongCount` делает ту же работу, что и `Count`, но возвращает 64-битное целочисленное значение, позволяя последовательностям содержать более 2 миллиардов элементов.

## Min и Max

| Аргумент | Тип |
|---|---|
| Входная последовательность | `IEnumerable<TSource>` |
| Селектор результатов (необязательный) | `TSource => TResult` |

Операции `Min` и `Max` возвращают наименьший или наибольший элемент из последовательности:

```
int[] numbers = { 28, 32, 14 };
int smallest = numbers.Min();  // 14;
int largest  = numbers.Max();  // 32;
```

Если указано выражение селектора, то каждый элемент сначала проецируется:

```
int smallest = numbers.Max (n => n % 10);  // 8;
```

Выражение селектора будет обязательным, если элементы сами по себе не являются внутренне сопоставимыми – другими словами, если они не реализуют интерфейс `IComparable<T>`:

```
Purchase runtimeError = dataContext.Purchases.Min ();           // Ошибка
decimal? lowestPrice = dataContext.Purchases.Min (p => p.Price); //Нормально
```

Выражение селектора определяет не только то, как элементы сравниваются, но также и финальный результат. В предыдущем примере финальным результатом оказывается десятичное значение, а не объект покупки. Для получения самой дешевой покупки понадобится подзапрос:

```
Purchase cheapest = dataContext.Purchases
  .Where (p => p.Price == dataContext.Purchases.Min (p2 => p2.Price))
  .FirstOrDefault();
```

В данном случае можно было бы сформулировать запрос без агрегации – используя операцию `OrderBy` и затем `FirstOrDefault`.

## Sum и Average

| Аргумент | Тип |
|---|---|
| Входная последовательность | `IEnumerable<TSource>` |
| Селектор результатов (необязательный) | `TSource => TResult` |

`Sum` и `Average` представляют собой операции агрегирования, которые применяются подобно `Min` и `Max`:

```
decimal[] numbers  = { 3, 4, 8 };
decimal sumTotal   = numbers.Sum();            // 15
decimal average    = numbers.Average();        // 5 (среднее значение)
```

Следующий запрос возвращает общую длину всех строк в массиве `names`:

```
int combinedLength = names.Sum (s => s.Length);// 19
```

Операции `Sum` и `Average` довольно ограничены в своей типизации. Их определения жестко привязаны к каждому числовому типу (`int`, `long`, `float`, `double`, `decimal`, а также их версии, допускающие `null`). Напротив, операции `Min` и `Max` могут напрямую оперировать на всех типах, которые реализуют интерфейс `IComparable<T>` – например, `string`.

Более того, операция `Average` всегда возвращает либо тип `decimal`, либо тип `double`, согласно следующей таблице.

| Тип селектора | Тип результата |
|---|---|
| `decimal` | `decimal` |
| `float` | `float` |
| `int, long, double` | `double` |

Это значит, что показанный ниже код не скомпилируется (будет выдано сообщение о невозможности преобразования double в int):

```
int avg = new int[] { 3, 4 }.Average();
```

Но следующий код скомпилируется:

```
double avg = new int[] { 3, 4 }.Average();  // 3.5
```

Операция Average неявно повышает входные значения, чтобы избежать потери точности. Выше в примере мы усредняем целочисленные значения и получаем 3.5 без необходимости в приведении входного элемента:

```
double avg = numbers.Average (n => (double) n);
```

При запрашивании базы данных операции Sum и Average транслируются в стандартные агрегации SQL. Представленный далее запрос возвращает заказчиков, у которых средняя покупка превышает сумму $500:

```
from c in dataContext.Customers
where c.Purchases.Average (p => p.Price) > 500
select c.Name;
```

## Aggregate

Операция Aggregate позволяет указывать специальный алгоритм накопления для реализации необычных агрегаций. Операция Aggregate не поддерживается в LINQ to SQL и Entity Framework, и сценарии ее использования стоят несколько особняком. Ниже показано, как с помощью Aggregate выполнить работу операции Sum:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate (0, (total, n) => total + n);  // 9
```

Первым аргументом операции Aggregate является *начальное значение*, с которого стартует накопление. Второй аргумент — выражение для обновления накопленного значения заданным новым элементом. Можно дополнительно предоставить третий аргумент, предназначенный для проецирования финального результирующего значения из накопленного значения.

Большинство задач, для которых была спроектирована операция Aggregate, можно легко решить с помощью цикла foreach — к тому же с применением более знакомого синтаксиса. Преимущество использования Aggregate заключается в том, что построение крупных или сложных агрегаций может быть автоматически распараллелено посредством PLINQ (глава 23).

### Агрегации без начального значения

При вызове операции Aggregate начальное значение может быть опущено; тогда первый элемент становится *неявным* начальным значением, и агрегация продолжается со второго элемента. Ниже приведен предыдущий пример *без использования начального значения*:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate ((total, n) => total + n);   // 6
```

Он дает тот же результат, что и ранее, но здесь выполняется *другое вычисление*. В предшествующем примере мы вычисляли 0+1+2+3, а теперь вычисляем 1+2+3. Лучше проиллюстрировать отличие поможет указание умножения вместо сложения:

```
int[] numbers = { 1, 2, 3 };
int x = numbers.Aggregate (0, (prod, n) => prod * n);  // 0*1*2*3 = 0
int y = numbers.Aggregate (   (prod, n) => prod * n);  //   1*2*3 = 6
```

Как будет показано в главе 23, агрегации без начального значения обладают преимуществом параллелизма, не требуя применения специальных перегруженных версий. Тем не менее, с такими агрегациями связан ряд проблем.

### Проблемы с агрегациями без начального значения

Методы агрегации без начального значения рассчитаны на использование с делегатами, которые являются *коммутативными* и *ассоциативными*. В случае их применения по-другому результат будет либо *непонятным* (в обычных запросах), либо *недетерминированным* (при распараллеливании запроса с помощью PLINQ). Например, рассмотрим следующую функцию:

```
(total, n) => total + n * n
```

Она не коммутативна и не ассоциативна. (Скажем, 1+2*2 != 2+1*1). Давайте посмотрим, что произойдет, если мы воспользуемся ею для суммирования квадратов чисел 2, 3 и 4:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate ((total, n) => total + n * n);  // 27
```

Вместо вычисления:

```
2*2 + 3*3 + 4*4       // 29
```

она вычисляет вот что:

```
2 + 3*3 + 4*4         // 27
```

Исправить ее можно несколькими способами. Для начала мы могли бы включить 0 в качестве первого элемента:

```
int[] numbers = { 0, 2, 3, 4 };
```

Это не только лишено элегантности, но будет еще и давать некорректные результаты при распараллеливании, поскольку PLINQ рассчитывает на ассоциативность функции, выбирая *несколько* элементов в качестве начальных. Чтобы проиллюстрировать сказанное, определим функцию агрегирования следующим образом:

```
f(total, n) => total + n * n
```

Инфраструктура LINQ to Objects вычислит ее так:

```
f(f(f(0, 2),3),4)
```

В то же время PLINQ может делать такое:

```
f(f(0,2),f(3,4))
```

с приведенным ниже результатом:

```
Первая часть:        a = 0 + 2*2  (= 4)
Вторая часть:        b = 3 + 4*4  (= 19)
Финальный результат:     a + b*b  (= 365)
ИЛИ ДАЖЕ ТАК:            b + a*a  (= 35)
```

Существуют два надежных решения. Первое – превратить функцию в агрегацию с нулевым начальным значением. Единственная сложность в том, что для PLINQ потребовалось бы использовать специальную перегруженную версию функции, чтобы

запрос не выполнялся последовательно (как объясняется в разделе "Оптимизация PLINQ" главы 23).

Второе решение предусматривает реструктуризацию запроса, так что функция агрегации становится коммутативной и ассоциативной:

```
int sum = numbers.Select (n => n * n).Aggregate ((total, n) => total + n);
```

Разумеется, в таких простых сценариях вы можете (и должны) применять операцию Sum вместо Aggregate:

```
int sum = numbers.Sum (n => n * n);
```

На самом деле с помощью только операций Sum и Average можно добиться довольно многого. Например, Average можно использовать для вычисления среднеквадратического значения:

```
Math.Sqrt (numbers.Average (n => n * n))
```

и даже стандартного отклонения:

```
double mean = numbers.Average();
double sdev = Math.Sqrt (numbers.Average (n =>
              {
                double dif = n - mean;
                return dif * dif;
              }));
```

Оба примера безопасны, эффективны и поддаются распараллеливанию. В главе 23 мы приведем практический пример специальной агрегации, которая не может быть сведена к Sum или Average.

# Квантификаторы

**IEnumerable<TSource> → *значение* bool**

| Метод | Описание | Эквиваленты в SQL |
|---|---|---|
| Contains | Возвращает true, если входная последовательность содержит заданный элемент | WHERE ... IN (...) |
| Any | Возвращает true, если любой элемент удовлетворяет заданному предикату | WHERE ... IN (...) |
| All | Возвращает true, если все элементы удовлетворяют заданному предикату | WHERE (...) |
| SequenceEqual | Возвращает true, если вторая последовательность содержит элементы, идентичные элементам в первой последовательности | |

## Contains и Any

Метод Contains принимает аргумент типа TSource, а Any — необязательный *предикат*.

Операция Contains возвращает true, если заданный элемент присутствует в последовательности:

```
bool hasAThree = new int[] { 2, 3, 4 }.Contains (3);  // true
```

Операция Any возвращает true, если указанное выражение дает значение true хотя бы для одного элемента. Предшествующий пример можно переписать с применением операции Any:

```
bool hasAThree = new int[] { 2, 3, 4 }.Any (n => n == 3);       // true
```

Операция Any может делать все, что делает Contains, и даже больше:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Any (n => n > 10);  // false
```

Вызов метода Any без предиката приводит к возвращению true, если последовательность содержит один или большее число элементов. Ниже показан другой способ записи предыдущего запроса:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Where (n => n > 10).Any();
```

Операция Any особенно удобна в подзапросах и часто используется при запрашивании баз данных, например:

```
from c in dataContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select c
```

### All и SequenceEqual

Операция All возвращает true, если все элементы удовлетворяют предикату. Следующий запрос возвращает заказчиков с покупками на сумму меньше $100:

```
dataContext.Customers.Where (c => c.Purchases.All (p => p.Price < 100));
```

Операция SequenceEqual сравнивает две последовательности. Для возвращения true обе последовательности должны иметь идентичные элементы, расположенные в одинаковом порядке. Можно дополнительно указать компаратор эквивалентности; по умолчанию применяется EqualityComparer<T>.Default.

## Методы генерации

**Ничего на входе → IEnumerable<TResult>**

| Метод | Описание |
|---|---|
| Empty | Создает пустую последовательность |
| Repeat | Создает последовательность повторяющихся элементов |
| Range | Создает последовательность целочисленных значений |

Empty, Repeat и Range являются статическими (не расширяющими) методами, которые создают простые локальные последовательности.

### Empty

Метод Empty создает пустую последовательность и требует только аргумента типа:

```
foreach (string s in Enumerable.Empty<string>())
  Console.Write (s);  // <ничего>
```

В сочетании с операцией ?? метод Empty выполняет действие, противоположное действию метода DefaultIfEmpty. Например, предположим, что имеется зубчатый

массив целых чисел, и требуется получить все целые числа в виде единственного плоского списка. Следующий запрос SelectMany терпит неудачу, если любой из внутренних массивов зубчатого массива оказывается null:

```
int[][] numbers =
{
  new int[] { 1, 2, 3 },
  new int[] { 4, 5, 6 },
  null  // Это значение null приводит к отказу запроса
};

IEnumerable<int> flat = numbers.SelectMany (innerArray => innerArray);
```

Проблема решается за счет использования комбинации метода Empty с операцией ??:

```
IEnumerable<int> flat = numbers
  .SelectMany (innerArray => innerArray ?? Enumerable.Empty <int>());

foreach (int i in flat)
  Console.Write (i + " ");     // 1 2 3 4 5 6
```

## Range и Repeat

Метод Range принимает начальный индекс и счетчик (оба значения являются целочисленными):

```
foreach (int i in Enumerable.Range (5, 3))
  Console.Write (i + " ");     // 5 6 7
```

Метод Repeat принимает элемент, подлежащий повторению, и количество повторений:

```
foreach (bool x in Enumerable.Repeat (true, 3))
  Console.Write (x + " ");     // True True True
```

# 10

# LINQ to XML

Платформа .NET Framework предоставляет несколько API-интерфейсов для работы с XML-данными. Начиная с версии .NET Framework 3.5, основным выбором для обработки универсальных XML-документов является *LINQ to XML*. Инфраструктура LINQ to XML состоит из легковесной, дружественной к LINQ объектной модели XML-документа и набора дополнительных операций запросов.

В настоящей главе мы сосредоточим внимание целиком на LINQ to XML. В главе 11 мы раскроем более специализированные XML-типы и API-интерфейсы, включая однонаправленные средства чтения/записи, типы для работы со схемами, таблицы стилей и XPath, а также унаследованную объектную модель документа (document object model – DOM), основанную на классе `XmlDocument`.

DOM-модель LINQ to XML исключительно хорошо спроектирована и отличается высокой производительностью. Даже без LINQ эта модель полезна в качестве легковесного фасада над низкоуровневыми классами `XmlReader` и `XmlWriter`.

Все типы LINQ to XML определены в пространстве имен `System.Xml.Linq`.

## Обзор архитектуры

Раздел начинается с очень краткого введения в концепции DOM-модели и продолжается объяснением логических обоснований, лежащих в основе DOM-модели LINQ to XML.

### Что собой представляет DOM-модель?

Рассмотрим следующее содержимое в XML-файле:

```
<?xml version="1.0" encoding="utf-8"?>
<customer id="123" status="archived">
  <firstname>Joe</firstname>
  <lastname>Bloggs</lastname>
</customer>
```

Как и все XML-файлы, он начинается с *объявления*, после которого следует корневой *элемент* по имени `customer`. Элемент `customer` имеет два *атрибута*, с каждым из которых связано имя (`id` и `status`) и значение (`"123"` и `"archived"`). Внутри `customer` присутствуют два дочерних элемента, `firstname` и `lastname`, каждый из которых имеет простое текстовое содержимое (`"Joe"` и `"Bloggs"`).

Каждая из упомянутых выше конструкций – объявление, элемент, атрибут, значение и текстовое содержимое – может быть представлена с помощью класса. А если такие классы имеют свойства коллекций для хранения дочернего содержимого, то для полного описания документа мы можем построить *дерево* объектов. Это и называется *объектной моделью документа*, или DOM-моделью.

## DOM-модель LINQ to XML

Инфраструктура LINQ to XML состоит из двух частей:

- DOM-модель XML, которую мы называем X-DOM;
- набор из примерно десятка дополнительных операций запросов.

Как и можно было ожидать, модель X-DOM состоит из таких типов, как `XDocument`, `XElement` и `XAttribute`. Интересно отметить, что типы X-DOM не привязаны к LINQ – модель X-DOM можно загружать, создавать экземпляры, обновлять и сохранять вообще без написания каких-либо запросов LINQ.

И наоборот, LINQ можно использовать для выдачи запросов к DOM-модели, созданной старыми типами, совместимыми с W3C. Однако такой подход утомителен и обладает ограниченными возможностями. Отличительной особенностью модели X-DOM является *дружественность к LINQ*, что означает следующее:

- она имеет методы, выпускающие удобные последовательности `IEnumerable`, которые можно запрашивать;
- ее конструкторы спроектированы так, что дерево X-DOM можно построить посредством проекции LINQ.

# Обзор модели X-DOM

На рис. 10.1 показаны основные типы модели X-DOM. Самым часто применяемым типом является `XElement`. Тип `XObject` представляет собой корень иерархии *наследования*, а типы `XElement` и `XDocument` – корни иерархии *включения*.

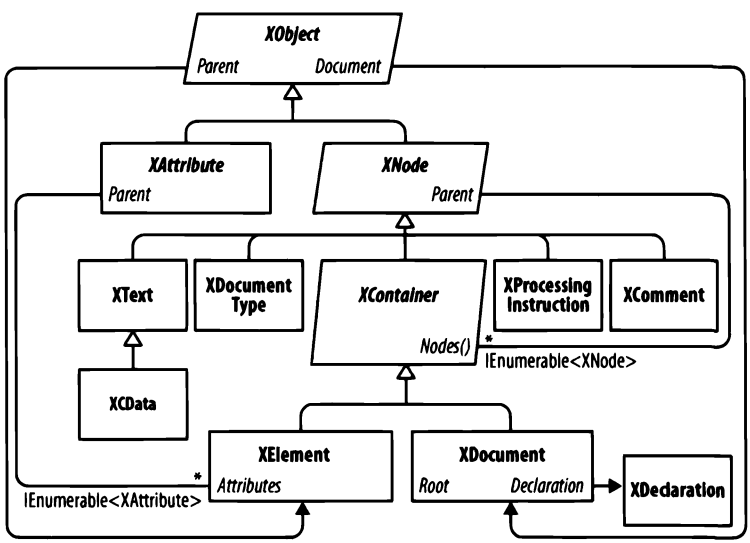



*Рис. 10.1. Основные типы X-DOM*

На рис. 10.2 изображено дерево X-DOM, созданное из приведенного ниже кода:

```
string xml = @"<customer id='123' status='archived'>
                <firstname>Joe</firstname>
                <lastname>Bloggs<!--nice name--></lastname>
              </customer>";
XElement customer = XElement.Parse (xml);
```

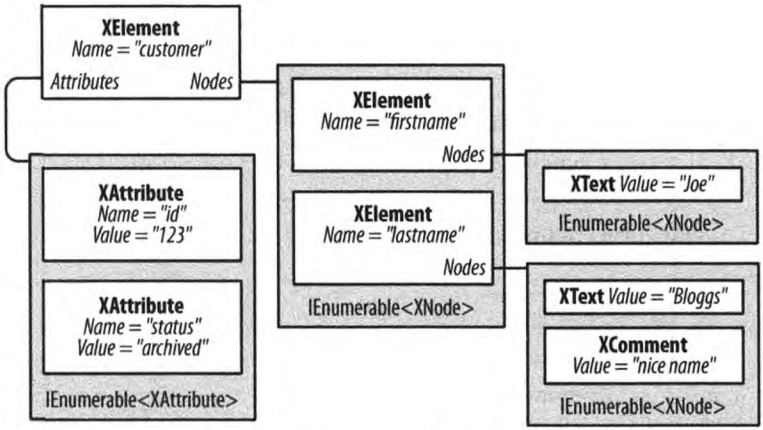



***Рис. 10.2.** Простое дерево X-DOM*

Тип `XObject` является абстрактным базовым классом для всего XML-содержимого. Он определяет ссылку на элемент `Parent` в контейнерном дереве, а также необязательный объект `XDocument`.

Тип `XNode` – базовый класс для большей части XML-содержимого, исключая атрибуты. Отличительная особенность объекта `XNode` в том, что он может находиться в упорядоченной коллекции смешанных типов `XNode`. Например, взгляните на такой XML-код:

```
<data>
  Hello world
  <subelement1/>
  <!--comment-->
  <subelement2/>
</data>
```

Внутри родительского элемента `<data>` сначала определен узел `XText` (`Hello world`), затем узел `XElement`, далее узел `XComment` и, наконец, еще один узел `XElement`. Напротив, объект `XAttribute` будет допускать в качестве равноправных узлов только другие объекты `XAttribute`.

Хотя `XNode` может обращаться к своему родительскому узлу `XElement`, концепция *дочерних* узлов в нем не предусмотрена: это забота его подклассов `XContainer`. Класс `XContainer` определяет члены для работы с дочерними узлами и является абстрактным базовым классом для `XElement` и `XDocument`.

В классе `XElement` определены члены для управления атрибутами, а также члены `Name` и `Value`. В (довольно распространенном) случае, когда элемент имеет единственный дочерний узел `XText`, свойство `Value` объекта `XElement` инкапсулирует содержимое этого дочернего узла для операций `get` и `set`, устраняя излишнюю навигацию. Благодаря `Value` можно по большей части избежать прямого взаимодействия с узлами `XText`.

Класс `XDocument` представляет корень XML-дерева. Выражаясь более точно, он *создает оболочку* для корневого узла `XElement`, добавляя объект `XDeclaration`, инструкции обработки и другие мелкие детали корневого уровня. В отличие от DOM-модели W3C использовать класс `XDocument` необязательно: вы можете загружать, манипулировать и сохранять модель X-DOM, даже не создавая объект `XDocument`! Необязательность `XDocument` также означает возможность эффективного и легкого перемещения поддерева узла в другую иерархию X-DOM.

## Загрузка и разбор

Классы `XElement` и `XDocument` предоставляют статические методы `Load` и `Parse`, предназначенные для построения дерева X-DOM из существующего источника:

- метод `Load` строит дерево X-DOM из файла, URI, объекта `Stream`, `TextReader` или `XmlReader`;
- метод `Parse` строит дерево X-DOM из строки.

Например:

```
XDocument fromWeb = XDocument.Load ("http://albahari.com/sample.xml");

XElement fromFile = XElement.Load (@"e:\media\somefile.xml");

XElement config = XElement.Parse (
@"<configuration>
   <client enabled='true'>
     <timeout>30</timeout>
   </client>
 </configuration>");
```

В последующих разделах мы покажем, каким образом выполнять обход и обновление дерева X-DOM. В качестве краткого обзора взгляните, как манипулировать только что наполненным элементом `config`:

```
foreach (XElement child in config.Elements())
  Console.WriteLine (child.Name);                    // client

XElement client = config.Element ("client");

bool enabled = (bool) client.Attribute ("enabled");  // Прочитать атрибут
Console.WriteLine (enabled);                         // True
client.Attribute ("enabled").SetValue (!enabled);    // Обновить атрибут

int timeout = (int) client.Element ("timeout");      // Прочитать элемент
Console.WriteLine (timeout);                         // 30
client.Element ("timeout").SetValue (timeout * 2);   // Обновить элемент

client.Add (new XElement ("retries", 3));        // Добавить новый элемент

Console.WriteLine (config);  // Неявно вызвать метод config.ToString
```

Вот результат последнего вызова `Console.WriteLine`:

```
<configuration>
  <client enabled="false">
    <timeout>60</timeout>
    <retries>3</retries>
  </client>
</configuration>
```

Класс XNode также предоставляет статический метод ReadFrom, который создает экземпляр любого типа узла и наполняет его из XmlReader. В отличие от Load он останавливается после чтения одного (полного) узла, так что затем можно вручную продолжить чтение из XmlReader.

Можно также делать обратное действие и применять XmlReader или XmlWriter для чтения или записи XNode через методы CreateReader и CreateWriter.

Мы опишем средства чтения и записи XML и объясним, как ими пользоваться, в главе 11.

## Сохранение и сериализация

Вызов метода ToString на любом узле преобразует его содержимое в XML-строку, сформатированную с разрывами и отступами, как только что было показано. (Разрывы строки и отступы можно запретить, указав SaveOptions.DisableFormatting при вызове ToString.)

Классы XElement и XDocument также предлагают метод Save, который записывает модель X-DOM в файл, объект Stream, TextWriter или XmlWriter. Если указан файл, то автоматически записывается и XML-объявление. В классе XNode также определен метод WriteTo, который принимает объект XmlWriter.

Мы более подробно опишем обработку XML-объявлений при сохранении в разделе "Документы и объявления" далее в главе.

# Создание экземпляра X-DOM

Вместо применения метода Load или Parse дерево X-DOM можно построить, вручную создавая объекты и добавляя их к родительскому узлу посредством метода Add класса XContainer.

Чтобы сконструировать объект XElement и XAttribute, нужно просто предоставить имя и значение:

```
XElement lastName = new XElement ("lastname", "Bloggs");
lastName.Add (new XComment ("nice name"));

XElement customer = new XElement ("customer");
customer.Add (new XAttribute ("id", 123));
customer.Add (new XElement ("firstname", "Joe"));
customer.Add (lastName);

Console.WriteLine (customer.ToString());
```

Вот результат:

```
<customer id="123">
  <firstname>Joe</firstname>
  <lastname>Bloggs<!--nice name--></lastname>
</customer>
```

При конструировании объекта XElement значение необязательно – можно указать только имя элемента, а содержимое добавить позже. Обратите внимание, что когда предоставляется значение, простой строки вполне достаточно – в явном создании и добавлении дочернего узла XText нет необходимости. Модель X-DOM выполняет эту работу автоматически, так что приходится иметь дело только со значениями.

## Функциональное построение

В предыдущем примере получить представление об XML-структуре на основании кода довольно-таки нелегко. Модель X-DOM поддерживает другой режим создания объектов, который называется *функциональным построением* (понятие, взятое из функционального программирования). При функциональном построении в единственном выражении строится целое дерево:

```
XElement customer =
  new XElement ("customer", new XAttribute ("id", 123),
    new XElement ("firstname", "joe"),
    new XElement ("lastname", "bloggs",
      new XComment ("nice name")
    )
  );
```

Такой подход обладает двумя преимуществами. Во-первых, код имеет сходство с формой результирующего кода XML. Во-вторых, он может быть включен в конструкцию `select` запроса LINQ. Например, следующий запрос LINQ to SQL проецируется прямо в дерево X-DOM:

```
XElement query =
  new XElement ("customers",
    from c in dataContext.Customers
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("firstname", c.FirstName),
        new XElement ("lastname", c.LastName,
          new XComment ("nice name")
        )
      )
  );
```

Более подробно об этом пойдет речь в разделе "Проецирование в модель X-DOM" далее в главе.

## Указание содержимого

Функциональное построение возможно из-за того, что конструкторы для `XElement` (и `XDocument`) перегружены с целью принятия массива типа `object[]` по имени `params`:

```
public XElement (XName name, params object[] content)
```

То же самое справедливо и в отношении метода `Add` в классе `XContainer`:

```
public void Add (params object[] content)
```

Таким образом, при построении или дополнении дерева X-DOM можно указывать любое количество дочерних объектов любых типов. Это работает, т.к. законным содержимым считается *все, что угодно*. Чтобы удостовериться в сказанном, необходимо посмотреть, как внутренне обрабатывается каждый объект содержимого. Ниже перечислены решения, которые по очереди принимает `XContainer`.

1. Если объект является `null`, то он игнорируется.
2. Если объект основан на `XNode` или `XStreamingElement`, тогда он добавляется в коллекцию `Nodes` в том виде, как есть.

3. Если объект является XAttribute, то он добавляется в коллекцию Attributes.
4. Если объект является строкой, тогда он помещается в узел XText и добавляется в коллекцию Nodes[1].
5. Если объект реализует интерфейс IEnumerable, то производится его перечисление с применением к каждому элементу тех же самых правил.
6. В противном случае объект преобразуется в строку, помещается в узел XText и затем добавляется в коллекцию Nodes[2].

В итоге все объекты попадают в одну из двух коллекций: Nodes или Attributes. Более того, любой объект является допустимым содержимым, потому что в конечном итоге на нем всегда можно вызывать метод ToString и трактовать как узел XText.

Перед вызовом метода ToString на произвольном типе реализация XContainer сначала проверяет, не относится ли он к одному из следующих типов:

```
float, double, decimal, bool,
DateTime, DateTimeOffset, TimeSpan
```

Если относится, тогда XContainer вызывает подходящим образом типизированный метод ToString на вспомогательном классе XmlConvert вместо вызова ToString на самом объекте. Это гарантирует, что данные поддерживают обмен и совместимы со стандартными правилами форматирования XML.

## Автоматическое глубокое копирование

Когда к элементу добавляется узел или атрибут (либо с помощью функционального построения, либо посредством метода Add), ссылка на данный элемент присваивается свойству Parent добавляемого узла или атрибута. Узел может иметь только один родительский элемент: если вы добавляете узел, уже имеющий родительский элемент, ко второму родительскому элементу, то этот узел автоматически подвергается *глубокому копированию*. В следующем примере каждый заказчик имеет отдельную копию address:

```
var address = new XElement ("address",
                new XElement ("street", "Lawley St"),
                new XElement ("town", "North Beach")
              );
var customer1 = new XElement ("customer1", address);
var customer2 = new XElement ("customer2", address);

customer1.Element ("address").Element ("street").Value = "Another St";
Console.WriteLine (
  customer2.Element ("address").Element ("street").Value);  // Lawley St
```

Такое автоматическое дублирование сохраняет создание объектов модели X-DOM свободным от побочных эффектов – еще один признак функционального программирования.

[1] В действительности модель X-DOM внутренне оптимизирует данный шаг, храня простое текстовое содержимое в строке. Узел XText фактически не создается вплоть до вызова метода Nodes на XContainer.

[2] См. сноску 1.

# Навигация и запросы

Как и можно было ожидать, в классах `XNode` и `XContainer` определены методы и свойства, предназначенные для обхода дерева X-DOM. Тем не менее, в отличие от обычной модели DOM такие методы и свойства не возвращают коллекцию, которая реализует интерфейс `IList<T>`. Взамен они возвращают либо одиночное значение, либо *последовательность*, реализующую интерфейс `IEnumerable<T>`, в отношении которой затем планируется выполнить запрос LINQ (или провести перечисление с помощью `foreach`). В результате появляется возможность запускать сложные запросы, а также решать простые задачи навигации с использованием знакомого синтаксиса запросов LINQ.

Имена элементов и атрибутов в X-DOM чувствительны к регистру – точно как в языке XML.

## Навигация по дочерним узлам

| Возвращаемый тип | Члены | С чем работают |
|---|---|---|
| `XNode` | `FirstNode { get; }` | `XContainer` |
| | `LastNode { get; }` | `XContainer` |
| `IEnumerable<XNode>` | `Nodes()` | `XContainer*` |
| | `DescendantNodes()` | `XContainer*` |
| | `DescendantNodesAndSelf()` | `XElement*` |
| `XElement` | `Element(XName)` | `XContainer` |
| `IEnumerable<XElement>` | `Elements()` | `XContainer*` |
| | `Elements(XName)` | `XContainer*` |
| | `Descendants()` | `XContainer*` |
| | `Descendants(XName)` | `XContainer*` |
| | `DescendantsAndSelf()` | `XElement*` |
| | `DescendantsAndSelf(XName)` | `XElement*` |
| `bool` | `HasElements { get; }` | `XElement` |

Функции, помеченные звездочкой (*) в третьей колонке в этой и других таблицах, также оперируют на *последовательностях* того же самого типа. Например, метод `Nodes` можно вызывать либо на объекте `XContainer`, либо на последовательности объектов `XContainer`. Такая возможность доступна благодаря расширяющим методам, которые определены в пространстве имен `System.Xml.Linq` – дополнительным операциям запросов, упомянутым в начале главы.

### `FirstNode`, `LastNode` и `Nodes`

Свойства `FirstNode` и `LastNode` предоставляют прямой доступ к первому и последнему дочернему узлу; метод `Nodes` возвращает все дочерние узлы в виде последовательности. Все три функции принимают во внимание только непосредственных потомков. Например:

```
var bench = new XElement ("bench",
            new XElement ("toolbox",
              new XElement ("handtool", "Hammer"),
              new XElement ("handtool", "Rasp")
            ),
            new XElement ("toolbox",
              new XElement ("handtool", "Saw"),
              new XElement ("powertool", "Nailgun")
            ),
            new XComment ("Be careful with the nailgun")
          );
foreach (XNode node in bench.Nodes())
  Console.WriteLine (node.ToString (SaveOptions.DisableFormatting) + ".");
```

Ниже показан вывод:

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>.
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>.
<!--Be careful with the nailgun-->.
```

## Извлечение элементов

Метод `Elements` возвращает только дочерние узлы типа `XElement`:

```
foreach (XElement e in bench.Elements())
  Console.WriteLine (e.Name + "=" + e.Value);     // toolbox=HammerRasp
                                                  // toolbox=SawNailgun
```

Следующий запрос LINQ находит ящик с пневматическим молотком (nail gun):

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  where toolbox.Elements().Any (tool => tool.Value == "Nailgun")
  select toolbox.Value;
```

```
РЕЗУЛЬТАТ: { "SawNailgun" }
```

В приведенном далее примере запрос `SelectMany` применяется для извлечения ручных инструментов (hand tool) из всех ящиков:

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  from tool in toolbox.Elements()
  where tool.Name == "handtool"
  select tool.Value;
```

```
РЕЗУЛЬТАТ: { "Hammer", "Rasp", "Saw" }
```

Сам по себе метод `Elements` является эквивалентом запроса LINQ в отношении `Nodes`. Предыдущий запрос можно было бы начать следующим образом:

```
from toolbox in bench.Nodes().OfType<XElement>()
where ...
```

Метод `Elements` может также возвращать только элементы с заданным именем. Например:

```
int x = bench.Elements ("toolbox").Count();  // 2
```

Данный код эквивалентен такому коду:

```
int x = bench.Elements().Where (e => e.Name == "toolbox").Count();  // 2
```

Кроме того, Elements определен как расширяющий метод, который принимает реализацию интерфейса IEnumerable<XContainer> или точнее аргумент следующего типа:

```
IEnumerable<T> where T : XContainer
```

Это позволяет ему работать также и с последовательностями элементов. С использованием метода Elements запрос, который ищет ручные инструменты во всех ящиках, можно записать так:

```
from tool in bench.Elements ("toolbox").Elements ("handtool")
select tool.Value.ToUpper();
```

Первый вызов Elements привязывается к методу экземпляра XContainer, а второй вызов Elements — к расширяющему методу.

### Извлечение одиночного элемента

Метод Element (с именем в форме единственного числа) возвращает первый совпадающий элемент с заданным именем. Метод Element удобен для простой навигации вроде продемонстрированной ниже:

```
XElement settings = XElement.Load ("databaseSettings.xml");
string cx = settings.Element ("database").Element ("connectString").Value;
```

Вызов Element эквивалентен вызову метода Elements с последующим применением операции запроса FirstOrDefault языка LINQ с предикатом сопоставления по имени. Метод Element возвращает null, если запрошенный элемент не существует.

Вызов Element("xyz").Value сгенерирует исключение NullReferenceException, когда элемент xyz не существует. Если вместо исключения предпочтительнее получить значение null, тогда вместо обращения к свойству Value необходимо привести XElement к типу string. Другими словами:

```
string xyz = (string) settings.Element ("xyz");
```

Прием работает из-за того, что в классе XElement определено явное преобразование в string, предназначенное как раз для такой цели!

Начиная с версии C# 6, доступна альтернатива — использование null-условной операции, т.е. Element("xyz")?.Value.

### Извлечение потомков

Класс XContainer также предлагает методы Descendants и DescendantNodes, которые возвращают дочерние элементы либо узлы вместе со всеми их дочерними элементами и т.д. (целое дерево). Метод Descendants принимает необязательное имя элемента. Возвращаясь к ранее рассмотренному примеру, вот как применить метод Descendants для поиска ручных инструментов:

```
Console.WriteLine (bench.Descendants ("handtool").Count()); // 3
```

Ниже продемонстрировано, что включаются и родительские, и листовые узлы:

```
foreach (XNode node in bench.DescendantNodes())
  Console.WriteLine (node.ToString (SaveOptions.DisableFormatting));
```

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>
<handtool>Hammer</handtool>
```

```
Hammer
<handtool>Rasp</handtool>
Rasp
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>
<handtool>Saw</handtool>
Saw
<powertool>Nailgun</powertool>
Nailgun
<!--Be careful with the nailgun-->
```

Следующий запрос извлекает из дерева X-DOM все комментарии, которые содержат слово "careful":

```
IEnumerable<string> query =
  from c in bench.DescendantNodes().OfType<XComment>()
  where c.Value.Contains ("careful")
  orderby c.Value
  select c.Value;
```

## Навигация по родительским узлам

Все классы XNode имеют свойство Parent и методы Ancestor*XXX*, предназначенные для навигации по родительским узлам. Родительский узел всегда представляет собой объект XElement.

| Возвращаемый тип | Члены | С чем работают |
|---|---|---|
| XElement | Parent { get; } | XNode* |
| Enumerable<XElement> | Ancestors()<br>Ancestors(XName)<br>AncestorsAndSelf()<br>AncestorsAndSelf(XName) | XNode*<br>XNode*<br>XElement*<br>XElement* |

Если x является XElement, тогда следующий код всегда выводит true:

```
foreach (XNode child in x.Nodes())
  Console.WriteLine (child.Parent == x);
```

Однако в случае, когда x представляет собой XDocument, все будет по-другому. Элемент XDocument особенный: он может иметь дочерние узлы, но никогда не может выступать родителем в отношении чего бы то ни было! Для доступа к XDocument должно использоваться свойство Document – оно работает на любом объекте в дереве X-DOM.

Метод Ancestors возвращает последовательность, первым элементом которой является Parent, следующим элементом – Parent.Parent и т.д. вплоть до корневого элемента.

Перейти к корневому элементу можно с помощью LINQ-запроса AncestorsAndSelf().Last. Другой способ достигнуть того же результата предусматривает обращение к свойству Document.Root, хотя такой прием работает только при наличии XDocument.

## Навигация по равноправным узлам

| Возвращаемый тип | Члены | Определены в |
|---|---|---|
| bool | IsBefore(XNode node) | XNode |
| | IsAfter(XNode node) | XNode |
| XNode | PreviousNode { get; } | XNode |
| | NextNode { get; } | XNode |
| IEnumerable<XNode> | NodesBeforeSelf() | XNode |
| | NodesAfterSelf() | XNode |
| IEnumerable<XElement> | ElementsBeforeSelf() | XNode |
| | ElementsBeforeSelf(XName name) | XNode |
| | ElementsAfterSelf() | XNode |
| | ElementsAfterSelf(XName name) | XNode |

С помощью свойств PreviousNode и NextNode (а также FirstNode/LastNode) узлы можно обходить с ощущением работы со связным списком. И это не случайно: внутренне узлы хранятся именно в связном списке.

Класс XNode внутренне применяет *односвязный* список, так что свойство PreviousNode не функционально.

## Навигация по атрибутам

| Возвращаемый тип | Члены | Определены в |
|---|---|---|
| bool | HasAttributes { get; } | XElement |
| XAttribute | Attribute(XName name) | XElement |
| | FirstAttribute { get; } | XElement |
| | LastAttribute { get; } | XElement |
| IEnumerable<XAttribute> | Attributes() | XElement |
| | Attributes(XName name) | XElement |

Вдобавок в XAttribute определены свойства PreviousAttribute и NextAttribute, а также Parent.

Метод Attributes, который принимает имя, возвращает последовательность с нулем или одним элементом; в XML элемент не может иметь дублированные имена атрибутов.

# Обновление модели X-DOM

Обновлять элементы и атрибуты можно следующими способами:

- вызвать метод SetValue или переустановить свойство Value;
- вызвать метод SetElementValue или SetAttributeValue;
- вызвать один из методов RemoveXXX;
- вызвать один из методов AddXXX или ReplaceXXX, указав новое содержимое.

Можно также переустанавливать свойство Name объектов XElement.

## Обновление простых значений

| Члены | С чем работают |
|---|---|
| `SetValue (object value)` | `XElement`, `XAttribute` |
| `Value { get; set }` | `XElement`, `XAttribute` |

Метод `SetValue` заменяет содержимое элемента или атрибута простым значением. Установка свойства `Value` делает то же самое, но принимает только строковые данные. Мы подробно опишем эти функции в разделе "Работа со значениями" далее в главе.

Эффект от вызова метода `SetValue` (или переустановки свойства `Value`) заключается в замене всех дочерних узлов:

```
XElement settings = new XElement ("settings",
                      new XElement ("timeout", 30)
                    );
settings.SetValue ("blah");
Console.WriteLine (settings.ToString());  // <settings>blah</settings>
```

## Обновление дочерних узлов и атрибутов

| Категория | Члены | С чем работают |
|---|---|---|
| Добавление | `Add (params object[] content)` | `XContainer` |
| | `AddFirst (params object[] content)` | `XContainer` |
| Удаление | `RemoveNodes()` | `XContainer` |
| | `RemoveAttributes()` | `XElement` |
| | `RemoveAll()` | `XElement` |
| Обновление | `ReplaceNodes (params object[] content)` | `XContainer` |
| | `ReplaceAttributes (params object[] content)` | `XElement` |
| | `ReplaceAll (params object[] content)` | `XElement` |
| | `SetElementValue (XName name, object value)` | `XElement` |
| | `SetAttributeValue (XName name, object value)` | `XElement` |

Наиболее удобными методами в данной группе являются последние два: `SetElementValue` и `SetAttributeValue`. Они служат сокращениями для создания экземпляра `XElement` или `XAttribute` и затем его добавления посредством `Add` к родительскому узлу с заменой любого существующего элемента или атрибута с таким же именем:

```
XElement settings = new XElement ("settings");
settings.SetElementValue ("timeout", 30);  // Добавляет дочерний узел
settings.SetElementValue ("timeout", 60);  // Обновляет его значением 60
```

Метод Add добавляет дочерний узел к элементу или документу. Метод AddFirst делает то же самое, но вставляет узел в начало коллекции, а не в ее конец.

С помощью метода RemoveNodes или RemoveAttributes можно удалить все дочерние узлы или атрибуты за один раз. Метод RemoveAll представляет собой эквивалент вызова обоих указанных методов.

Методы Replace*XXX* являются эквивалентами вызова сначала Remove*XXX*, а затем Add*XXX*. Они получают копию входных данных, так что e.ReplaceNodes(e.Nodes()) работает ожидаемым образом.

## Обновление через родительский элемент

| Члены | С чем работают |
|---|---|
| AddBeforeSelf (params object[] content) | XNode |
| AddAfterSelf (params object[] content) | XNode |
| Remove() | XNode*, XAttribute* |
| ReplaceWith (params object[] content) | XNode |

Методы AddBeforeSelf, AddAfterSelf, Remove и ReplaceWith не оперируют на дочерних узлах заданного узла. Взамен они работают с коллекцией, в которой находится сам узел. Это требует, чтобы узел имел родительский элемент – иначе сгенерируется исключение. Методы AddBeforeSelf и AddAfterSelf удобны для вставки узла в произвольную позицию:

```
XElement items = new XElement ("items",
                 new XElement ("one"),
                 new XElement ("three")
                );
items.FirstNode.AddAfterSelf (new XElement ("two"));
```

Ниже показан результат:

```
<items><one /><two /><three /></items>
```

Вставка в произвольную позицию внутри длинной последовательности элементов на самом деле довольно эффективна, поскольку внутренне узлы хранятся в связном списке.

Метод Remove удаляет текущий узел из его родительского узла. Метод ReplaceWith делает то же самое, но затем вставляет в ту же самую позицию другое содержимое. Например:

```
XElement items = XElement.Parse ("<items><one/><two/><three/></items>");
items.FirstNode.ReplaceWith (new XComment ("one was here"));
```

Вот результат:

```
<items><!--one was here--><two /><three /></items>
```

### Удаление последовательности узлов или атрибутов

Благодаря расширяющим методам из пространства имен System.Xml.Linq метод Remove можно также вызывать на *последовательности* узлов или атрибутов. Взгляните на следующую модель X-DOM:

```
XElement contacts = XElement.Parse (
@"<contacts>
   <customer name='Mary'/>
   <customer name='Chris' archived='true'/>
   <supplier name='Susan'>
     <phone archived='true'>012345678<!--confidential--></phone>
   </supplier>
 </contacts>");
```

Приведенный ниже вызов удаляет всех заказчиков:

```
contacts.Elements ("customer").Remove();
```

Следующий оператор удаляет все архивные ("archived") контакты (так что запись *Chris* больше не будет видна):

```
contacts.Elements().Where (e => (bool?) e.Attribute ("archived") == true)
                   .Remove();
```

Если мы заменим вызов метода `Elements` вызовом `Descendants`, то все архивные элементы в DOM-модели больше не будут видны, и результат окажется следующим:

```
<contacts>
  <customer name="Mary" />
  <supplier name="Susan" />
</contacts>
```

В показанном ниже примере удаляются все контакты, которые имеют комментарий "confidential" (конфиденциально) в любом месте своего дерева:

```
contacts.Elements().Where (e => e.DescendantNodes()
                                 .OfType<XComment>()
                                 .Any (c => c.Value == "confidential")
                          ).Remove();
```

Результат будет таким:

```
<contacts>
  <customer name="Mary" />
  <customer name="Chris" archived="true" />
</contacts>
```

Сравните это со следующим более простым запросом, который удаляет все узлы комментариев из дерева:

```
contacts.DescendantNodes().OfType<XComment>().Remove();
```

Внутренне методы `Remove` сначала помещают все совпадающие элементы во временный список, после чего выполняют его перечисление, чтобы произвести удаления. Такой подход позволяет избежать ошибок, которые могут в противном случае возникнуть из-за удаления и запрашивания в один и тот же момент.

# Работа со значениями

В классах `XElement` и `XAttribute` определено свойство `Value` типа `string`. Если элемент имеет единственный дочерний узел `XText`, тогда свойство `Value` класса `XElement` действует в качестве удобного сокращения для доступа к содержимому такого узла. В случае класса `XAttribute` свойство `Value` — это просто значение атрибута.

Несмотря на отличия в хранении, модель X-DOM предоставляет согласованный набор операций для работы со значениями элементов и атрибутов.

## Установка значений

Существуют два способа присваивания значения: вызов метода `SetValue` или установка свойства `Value`. Метод `SetValue` гибче, т.к. он принимает не только строки, но и другие простые типы данных:

```
var e = new XElement ("date", DateTime.Now);
e.SetValue (DateTime.Now.AddDays(1));
Console.Write (e.Value);  // 2018-03-02T16:39:10.734375+09:00
```

Мы бы могли взамен просто установить свойство `Value` элемента, но тогда пришлось бы вручную преобразовывать значение `DateTime` в строку. Такое действие сложнее обычного вызова метода `ToString`, потому что требует использования класса `XmlConvert` для получения результата, совместимого с XML.

Когда конструктору класса `XElement` или `XAttribute` передается *значение*, то же самое автоматическое преобразование выполняется для нестроковых типов. В результате гарантируется корректность форматирования значений `DateTime`; значение `true` записывается в нижнем регистре, а `double.NegativeInfinity` записывается в виде `-INF`.

## Получение значений

Чтобы пойти другим путем и разобрать значение `Value` обратно в базовый тип, необходимо лишь привести `XElement` или `XAttribute` к желаемому типу. Прием выглядит так, как будто он не должен работать — но он работает! Например:

```
XElement e = new XElement ("now", DateTime.Now);
DateTime dt = (DateTime) e;

XAttribute a = new XAttribute ("resolution", 1.234);
double res = (double) a;
```

Элемент или атрибут не хранит значения `DateTime` или числа в их естественной форме — они всегда хранятся в виде текста и при необходимости разбираются. Кроме того, исходный тип не запоминается, поэтому приводить нужно корректно, чтобы избежать ошибки во время выполнения. Для повышения надежности кода приведение можно поместить в блок `try/catch`, перехватывающий исключение `FormatException`.

Явные приведения `XElement` и `XAttribute` могут обеспечить разбор в следующие типы:

- все стандартные числовые типы;
- типы `string`, `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan` и `Guid`;
- версии `Nullable<>` вышеупомянутых типов значений.

Приведение к типу, допускающему `null`, удобно применять в сочетании с методами `Element` и `Attribute`, т.к. даже если запрошенное имя не существует, то приведение все равно работает. Например, если `x` не имеет элемента `timeout`, тогда первая строка сгенерирует ошибку во время выполнения, а вторая — нет:

```
int timeout = (int) x.Element ("timeout");   // Ошибка
int? timeout = (int?) x.Element ("timeout"); //Нормально; timeout равно null
```

С помощью операции ?? в финальном результате можно избавиться от типа, допускающего null. Если атрибут resolution не существует, то переменная resolution получит значение 1.0:

```
double resolution = (double?) x.Attribute ("resolution") ?? 1.0;
```

Тем не менее, приведение к типу, допускающему null, не избавит от неприятностей, если элемент или атрибут *существует* и имеет пустое (либо неправильно сформатированное) значение. Для этого придется перехватывать исключение FormatException.

Приведения можно также использовать в запросах LINQ. Представленный ниже запрос возвращает John:

```
var data = XElement.Parse (
  @"<data>
      <customer id='1' name='Mary' credit='100' />
      <customer id='2' name='John' credit='150' />
      <customer id='3' name='Anne' />
    </data>");

IEnumerable<string> query = from cust in data.Elements()
                            where (int?) cust.Attribute ("credit") > 100
                            select cust.Attribute ("name").Value;
```

Приведение к типу int, допускающему null, позволяет избежать исключения NullReferenceException для заказчика Anne, у которого отсутствует атрибут credit. Другое решение могло бы предусматривать добавление предиката в конструкцию where:

```
where cust.Attributes ("credit").Any() && (int) cust.Attribute...
```

Те же самые принципы применяются при запрашивании значений элементов.

## Значения и узлы со смешанным содержимым

Имея доступ к значению Value, может возникнуть вопрос, нужно ли вообще работать напрямую с узлами XText? Да, если они имеют смешанное содержимое. Например:

```
<summary>An XAttribute is <bold>not</bold> an XNode</summary>
```

Простого свойства Value для захвата содержимого summary недостаточно. В элементе summary присутствуют три дочерних узла: XText, XElement и XText. Вот как их сконструировать:

```
XElement summary = new XElement ("summary",
                     new XText ("An XAttribute is "),
                     new XElement ("bold", "not"),
                     new XText (" an XNode")
                   );
```

Интересно отметить, что мы по-прежнему можем обращаться к свойству Value элемента summary без генерации исключения. Взамен мы получаем конкатенацию значений всех дочерних узлов:

```
An XAttribute is not an XNode
```

Также разрешено переустанавливать свойство Value элемента summary ценой замены всех предыдущих дочерних узлов единственным новым узлом XText.

### Автоматическая конкатенация XText

При добавлении простого содержимого в XElement модель X-DOM дополняет существующий дочерний узел XText, а не создает новый. В следующих примерах e1 и e2 получают только один дочерний элемент XText со значением HelloWorld:

```
var e1 = new XElement ("test", "Hello"); e1.Add ("World");
var e2 = new XElement ("test", "Hello", "World");
```

Однако если узлы XText создаются специально, тогда дочерних элементов будет несколько:

```
var e = new XElement ("test", new XText ("Hello"), new XText ("World"));
Console.WriteLine (e.Value);                // HelloWorld
Console.WriteLine (e.Nodes().Count());      // 2
```

Объект XElement не выполняет конкатенацию двух узлов XText, поэтому идентичности объектов узлов предохраняются.

## Документы и объявления

### XDocument

Как упоминалось ранее, объект XDocument является оболочкой для корневого элемента XElement и позволяет добавлять элемент XDeclaration, инструкции обработки, тип документа и комментарии корневого уровня. Объект XDocument не является обязательным и может быть проигнорирован или опущен: в отличие от DOM-модели W3C он не служит средством объединения всего вместе.

Класс XDocument предлагает те же самые функциональные конструкторы, что и класс XElement. И поскольку он основан на XContainer, в нем также поддерживаются методы Add*XXX*, Remove*XXX* и Replace*XXX*. Тем не менее, в отличие от XElement класс XDocument может принимать только ограниченное содержимое:

- единственный объект XElement ("корень");
- единственный объект XDeclaration;
- единственный объект XDocumentType (для ссылки на DTD);
- любое количество объектов XProcessingInstruction;
- любое количество объектов XComment.

Для получения допустимого объекта XDocument из всего перечисленного обязательным считается только корневой объект XElement. Объект XDeclaration необязателен – при его отсутствии во время сериализации применяются стандартные настройки.

Простейший допустимый XDocument имеет только корневой элемент:

```
var doc = new XDocument (
            new XElement ("test", "data")
          );
```

Обратите внимание, что мы не включили объект XDeclaration. Однако файл, созданный в результате вызова метода doc.Save, будет по-прежнему содержать XML-объявление, потому что оно генерируется по умолчанию.

В следующем примере создается простой, но корректный XHTML-файл, иллюстрирующий все конструкции, которые может принимать XDocument:

```
var styleInstruction = new XProcessingInstruction (
  "xml-stylesheet", "href='styles.css' type='text/css'");

var docType = new XDocumentType ("html",
  "-//W3C//DTD XHTML 1.0 Strict//EN",
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd", null);

XNamespace ns = "http://www.w3.org/1999/xhtml";
var root =
  new XElement (ns + "html",
    new XElement (ns + "head",
      new XElement (ns + "title", "An XHTML page")),
    new XElement (ns + "body",
      new XElement (ns + "p", "This is the content"))
  );

var doc =
  new XDocument (
    new XDeclaration ("1.0", "utf-8", "no"),
    new XComment ("Reference a stylesheet"),
    styleInstruction,
    docType,
    root);

doc.Save ("test.html");
```

Ниже показано содержимое результирующего файла *test.html*:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!--Reference a stylesheet-->
<?xml-stylesheet href='styles.css' type='text/css'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>An XHTML page</title>
  </head>
  <body>
    <p>This is the content</p>
  </body>
</html>
```

Класс XDocument имеет свойство Root, которое служит сокращением для доступа к единственному объекту XElement документа. Обратная ссылка предоставляется свойством Document класса XObject, которая работает для всех объектов в дереве:

```
Console.WriteLine (doc.Root.Name.LocalName);    // html
XElement bodyNode = doc.Root.Element (ns + "body");
Console.WriteLine (bodyNode.Document == doc);   // True
```

Вспомните, что дочерние узлы документа не имеют родительского элемента:

```
Console.WriteLine (doc.Root.Parent == null);    // True
foreach (XNode node in doc.Nodes())
  Console.Write (node.Parent == null);          // TrueTrueTrueTrue
```

Объект `XDeclaration` — это не `XNode` и в отличие от комментариев, инструкций обработки и корневого элемента он не должен присутствовать внутри коллекции `Nodes` документа. Взамен он присваивается отдельному свойству по имени `Declaration`. Именно потому в последнем примере значение `True` в выводе повторялось четыре раза, а не пять.

## Объявления XML

Стандартный XML-файл начинается с примерно такого объявления:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

Объявление XML гарантирует, что содержимое файла будет корректно разобрано и воспринято средством чтения. При выдаче объявлений XML объекты `XElement` и `XDocument` следуют описанным ниже правилам:

- вызов метода `Save` с именем файла всегда записывает объявление;
- вызов метода `Save` с экземпляром `XmlWriter` записывает объявление, если только `XmlWriter` не был проинструктирован иначе;
- метод `ToString` никогда не выпускает объявление XML.

Объект `XmlWriter` можно инструктировать о том, что он не должен генерировать объявление, путем установки свойств `OmitXmlDeclaration` и `ConformanceLevel` объекта `XmlWriterSettings` при конструировании экземпляра `XmlWriter`. Об этом пойдет речь в главе 11.

Наличие или отсутствие объекта `XDeclaration` никак не влияет на то, записывается объявление XML либо нет. Объект `XDeclaration` предназначен для *предоставления подсказок XML-сериализации* двумя способами:

- какую кодировку текста использовать;
- что именно помещать в атрибуты `encoding` и `standalone` объявления XML (должны записываться объявлением).

Конструктор класса `XDeclaration` принимает три аргумента, которые соответствуют атрибутам `version`, `encoding` и `standalone`. В следующем примере содержимое *`test.xml`* кодируется с применением UTF-16:

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-16", "yes"),
            new XElement ("test", "data")
          );
doc.Save ("test.xml");
```

Что бы ни было указано для версии XML, средство записи XML его игнорирует, всегда записывая `"1.0"`.

Кодировка должна указываться с использованием кода IETF, подобного `"utf-16"` — в точности, как он будет представлен в объявлении XML.

### Запись объявления в строку

Предположим, что объект `XDocument` необходимо сериализировать в строку, включая объявление XML. Поскольку метод `ToString` не записывает объявление, мы должны применять вместо него `XmlWriter`:

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-8", "yes"),
            new XElement ("test", "data")
          );
var output = new StringBuilder();
var settings = new XmlWriterSettings { Indent = true };
using (XmlWriter xw = XmlWriter.Create (output, settings))
  doc.Save (xw);
Console.WriteLine (output.ToString());
```

Вот результат:

```
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<test>data</test>
```

Обратите внимание, что в выводе получена кодировка UTF-16 — несмотря на то, что в `XDeclaration` была явно затребована кодировка UTF-8! Результат может выглядеть как ошибка, но на самом деле объект `XmlWriter` удивительно интеллектуален. Из-за того, что запись производится в строку, а не в файл или поток, невозможно использовать никакую другую кодировку кроме UTF-16 — формат, в котором внутренне хранятся строки. Таким образом, `XmlWriter` записывает `"utf-16"` — так что никакого обмана здесь нет.

Это также объясняет причину, по которой метод `ToString` не выпускает объявление XML. Представьте, что вместо вызова метода `Save` для записи `XDocument` в файл вы поступаете следующим образом:

```
File.WriteAllText ("data.xml", doc.ToString());
```

Как уже утверждалось, в файле *`data.xml`* будет отсутствовать объявление XML, делая его незавершенным, однако по-прежнему поддающимся разбору (кодировка текста может быть выведена). Но если бы метод `ToString` выдавал объявление XML, тогда файл *`data.xml`* фактически содержал бы *некорректное* объявление (`encoding="utf-16"`), которое могло бы препятствовать его успешному чтению, потому что метод `WriteAllText` кодирует с применением UTF-8.

# Имена и пространства имен

Точно так же как типы .NET могут иметь пространства имен, то же самое возможно для элементов и атрибутов XML.

Пространства имен XML преследуют две цели. Во-первых, подобно пространствам имен в языке C# они помогают избегать конфликтов имен. Такая проблема может возникать при слиянии данных из нескольких XML-файлов. Во-вторых, пространства имен придают имени *абсолютный* смысл. Например, имя `nil` может означать все что угодно. Тем не менее, в рамках пространства имен *`http://www.w3.org/2001/xmlschema-instance`* имя `nil` означает своего рода эквивалент значения `null` в C# и сопровождается специфичными правилами его использования.

Поскольку пространства имен XML являются существенным источником путаницы, мы раскроем данную тему сначала в общих чертах и затем перейдем к применению пространств имен в LINQ to XML.

## Пространства имен в XML

Предположим, что требуется определить элемент `customer` в пространстве имен `OReilly.Nutshell.CSharp`. Сделать это можно двумя способами. Первый способ – воспользоваться атрибутом `xmlns`:

```
<customer xmlns="OReilly.Nutshell.CSharp"/>
```

`xmlns` представляет собой специальный зарезервированный атрибут. В случае применения в подобной манере он выполняет две функции:

- указывает пространство имен для данного элемента;
- указывает стандартное пространство имен для всех элементов-потомков.

Таким образом, в следующем примере `address` и `postcode` неявно находятся в пространстве имен `OReilly.Nutshell.CSharp`:

```
<customer xmlns="OReilly.Nutshell.CSharp">
  <address>
    <postcode>02138</postcode>
  </address>
</customer>
```

Если нужно, чтобы `address` и `postcode` *не* имели пространства имен, тогда понадобится поступить так:

```
<customer xmlns="OReilly.Nutshell.CSharp">
  <address xmlns="">
    <postcode>02138</postcode>  <!-- postcode теперь наследует пустое
                                пространство имен -->
  </address>
</customer>
```

### Префиксы

Другой способ указания пространства имен предусматривает использование *префикса*. Префикс – это псевдоним, который назначается пространству имен с целью сокращения клавиатурного ввода. С применением префикса связаны два шага – *определение* префикса и его *использование*. Шаги можно объединить:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp"/>
```

Здесь происходят два разных действия. В правой части конструкция `xmlns:nut="..."` определяет префикс по имени `nut` и делает его доступным данному элементу и всем его потомкам. В левой части конструкция `nut:customer` назначает вновь выделенный префикс элементу `customer`.

Элемент, снабженный префиксом, *не* определяет стандартное пространство имен для потомков. В следующем XML-коде `firstname` имеет пустое пространство имен:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp">
  <firstname>Joe</firstname>
</customer>
```

Чтобы назначить `firstname` пространство имен `OReilly.Nutshell.CSharp`, потребуется поступить так:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp">
  <nut:firstname>Joe</firstname>
</customer>
```

Префикс – или префиксы – можно также определять для удобства работы с потомками, не назначая любой из этих префиксов самому родительскому элементу. В показанном ниже коде определены два префикса, i и z, а сам элемент customer оставлен с пустым пространством имен:

```
<customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  ...
</customer>
```

Если бы узел был корневым, то i и z были бы доступны всему документу. Префиксы удобны, когда элементы должны извлекаться из нескольких пространств имен.

Обратите внимание, что в рассмотренном примере оба пространства имен представляют собой URI. Применение URI (которыми вы владеете) является стандартной практикой: оно обеспечивает уникальность пространств имен. Таким образом, в реальных обстоятельствах элемент customer, скорее всего, будет больше похож на:

```
<customer xmlns="http://oreilly.com/schemas/nutshell/csharp"/>
```

или на:

```
<nut:customer xmlns:nut="http://oreilly.com/schemas/nutshell/csharp"/>
```

### Атрибуты

Назначать пространства имен можно также и атрибутам. Главное отличие состоит в том, что атрибут всегда требует префикса. Например:

```
<customer xmlns:nut="OReilly.Nutshell.CSharp" nut:id="123" />
```

Еще одно отличие связано с тем, что неуточненный атрибут всегда имеет пустое пространство имен: он никогда не наследует стандартное пространство имен от своего родительского элемента.

Атрибуты, как правило, не нуждаются в пространствах имен, потому что их смысл обычно локален по отношению к элементу. Исключение составляют универсальные атрибуты или атрибуты метаданных, такие как атрибут nil, определенный W3C:

```
<customer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstname>Joe</firstname>
  <lastname xsi:nil="true"/>
</customer>
```

Здесь однозначно указано на то, что lastname является nil (null в C#), а не пустой строкой. Поскольку мы используем стандартное пространство имен, универсальная утилита разбора может точно определить наше намерение.

## Указание пространств имен в X-DOM

До сих пор в настоящей главе в качестве имен XElement и XAttribute мы применяли только простые строки. Простая строка соответствует имени XML с пустым пространством имен – очень похоже на тип .NET, определенный в глобальном пространстве имен.

Существует пара подходов к указанию пространства имен XML. Первый из них – заключение его в фигурные скобки и помещение перед локальным именем. Например:

```
var e = new XElement ("{http://domain.com/xmlspace}customer", "Bloggs");
Console.WriteLine (e.ToString());
```

Вот результирующий XML-код:

```
<customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

Второй (и более производительный) подход предусматривает использование типов XNamespace и XName. Их определения показаны ниже:

```
public sealed class XNamespace
{
  public string NamespaceName { get; }
}

public sealed class XName //Локальное имя с дополнительным пространством имен
{
  public string LocalName { get; }
  public XNamespace Namespace { get; }  // Необязательно
}
```

Оба типа определяют неявные приведения от string, поэтому следующий код вполне законен:

```
XNamespace ns   = "http://domain.com/xmlspace";
XName localName = "customer";
XName fullName  = "{http://domain.com/xmlspace}customer";
```

В типе XNamespace также перегружена операция +, что позволяет комбинировать пространство имен с именем в XName, не применяя фигурные скобки:

```
XNamespace ns = "http://domain.com/xmlspace";
XName fullName = ns + "customer";
Console.WriteLine (fullName);  // {http://domain.com/xmlspace}customer
```

Все конструкторы и методы в модели X-DOM, которые принимают имя элемента или атрибута, на самом деле принимают объект XName, а не строку. Причина того, что строку можно заменять (как во всех примерах, приведенных до настоящего момента), связана с неявным приведением.

Независимо от того, чем является сущность — элементом или атрибутом — пространство имен указывается одинаково:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XAttribute (ns + "id", 123)
           );
```

## Модель X-DOM и стандартные пространства имен

Модель X-DOM игнорирует концепцию стандартного пространства имен до тех пор, пока не наступает момент фактического вывода XML. Это означает, что когда вы конструируете дочерний элемент XElement, то при необходимости должны предоставить его пространство имен явно: оно *не будет* наследоваться от родительского элемента:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer", "Bloggs"),
             new XElement (ns + "purchase", "Bicycle")
           );
```

Однако при чтении и выводе XML модель X-DOM использует стандартные пространства имен:

```
Console.WriteLine (data.ToString());
```

ВЫВОД:

```
<data xmlns="http://domain.com/xmlspace">
  <customer>Bloggs</customer>
  <purchase>Bicycle</purchase>
</data>
```

```
Console.WriteLine (data.Element (ns + "customer").ToString());
```

ВЫВОД:

```
<customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

Если дочерние узлы XElement конструируются без указания пространств имен — другими словами, так:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement ("customer", "Bloggs"),
             new XElement ("purchase", "Bicycle")
           );
Console.WriteLine (data.ToString());
```

тогда будет получен отличающийся результат:

```
<data xmlns="http://domain.com/xmlspace">
  <customer xmlns="">Bloggs</customer>
  <purchase xmlns="">Bicycle</purchase>
</data>
```

Еще одна проблема возникает, когда по причине забывчивости не включить пространство имен при навигации по дереву X-DOM:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer", "Bloggs"),
             new XElement (ns + "purchase", "Bicycle")
           );
XElement x = data.Element (ns + "customer");     // Нормально
XElement y = data.Element ("customer");          // null
```

Если вы строите дерево X-DOM, не указывая пространства имен, то можете впоследствии назначить любому элементу одиночное пространство имен, как показано ниже:

```
foreach (XElement e in data.DescendantsAndSelf())
  if (e.Name.Namespace == "")
    e.Name = ns + e.Name.LocalName;
```

## Префиксы

Модель X-DOM трактует префиксы точно так же, как пространства имен: чисто как функцию сериализации. Следовательно, вы можете полностью игнорировать проблему префиксов — и это сойдет вам с рук! Единственная причина, по которой вы можете решить поступить иначе, касается эффективности при выводе в XML-файл. Например, взгляните на приведенный далее код:

```
XNamespace ns1 = "http://domain.com/space1";
XNamespace ns2 = "http://domain.com/space2";
var mix = new XElement (ns1 + "data",
          new XElement (ns2 + "element", "value"),
          new XElement (ns2 + "element", "value"),
          new XElement (ns2 + "element", "value")
        );
```

По умолчанию XElement будет сериализировать результат следующим образом:

```
<data xmlns="http://domain.com/space1">
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
</data>
```

Как видите, присутствует излишнее дублирование. Решение заключается в том, чтобы *не* изменять способ конструирования X-DOM, а взамен предоставить сериализатору подсказки перед записью XML. Для этого необходимо добавить атрибуты, определяющие префиксы, которые должны быть применены. Обычно такие атрибуты добавляются к корневому элементу:

```
mix.SetAttributeValue (XNamespace.Xmlns + "ns1", ns1);
mix.SetAttributeValue (XNamespace.Xmlns + "ns2", ns2);
```

Приведенный выше код назначает префикс ns1 переменной ns1 из XNamespace и префикс ns2 – переменной ns2. Во время сериализации модель X-DOM автоматически выбирает указанные атрибуты и использует их для уплотнения результирующего XML. Вот результат вызова метода ToString на mix:

```
<ns1:data xmlns:ns1="http://domain.com/space1"
          xmlns:ns2="http://domain.com/space2">
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
</ns1:data>
```

Префиксы не изменяют способа конструирования, запрашивания или обновления модели X-DOM – для таких действий мы игнорируем наличие префиксов и продолжаем применять полные имена. Префиксы вступают в игру только при преобразовании в и из файлов или потоков XML.

Префиксы также учитываются в атрибутах сериализации. В приведенном ниже примере мы записываем дату рождения и кредит заказчика в виде "nil", используя стандартный атрибут W3C. Выделенная строка гарантирует, что префикс сериализируется без нежелательного повторения пространства имен:

```
XNamespace xsi = "http://www.w3.org/2001/XMLSchema-instance";
var nil = new XAttribute (xsi + "nil", true);

var cust = new XElement ("customers",
           new XAttribute (XNamespace.Xmlns + "xsi", xsi),
           new XElement ("customer",
             new XElement ("lastname", "Bloggs"),
             new XElement ("dob", nil),
             new XElement ("credit", nil)
           )
         );
```

А вот результирующий XML-код:

```
<customers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <customer>
    <lastname>Bloggs</lastname>
    <dob xsi:nil="true" />
    <credit xsi:nil="true" />
  </customer>
</customers>
```

Для краткости мы предварительно объявили пустой XAttribute, так что его можно применять два раза при построении DOM-модели. Дважды ссылаться на тот же самый атрибут разрешено из-за того, что при необходимости он автоматически дублируется.

# Аннотации

С помощью аннотации к любому объекту XObject можно присоединять специальные данные. Аннотации предназначены для вашего личного использования и трактуются моделью X-DOM как черные ящики. Если вы когда-либо имели дело со свойством Tag элемента управления Windows Forms или WPF, то концепция должна быть знакомой – лишь с тем отличием, что разрешено иметь множество аннотаций, и назначать им *закрытую область видимости*. Допускается создать аннотацию, которую другие типы не смогут даже видеть, не говоря уже о том, чтобы перезаписывать.

За добавление и удаление аннотаций отвечают следующие методы класса XObject:

```
public void AddAnnotation (object annotation)
public void RemoveAnnotations<T>()     where T : class
```

Перечисленные далее методы извлекают аннотации:

```
public T Annotation<T>()               where T : class
public IEnumerable<T> Annotations<T>() where T : class
```

Каждой аннотации назначается ключ согласно ее *типу*, который должен быть ссылочным. Показанный ниже код добавляет и затем извлекает аннотацию типа string:

```
XElement e = new XElement ("test");
e.AddAnnotation ("Hello");
Console.WriteLine (e.Annotation<string>());  // Hello
```

Можно добавить множество аннотаций того же самого типа, а затем применить метод Annotations для извлечения *последовательности* совпадений.

Тем не менее, открытый тип вроде string не обеспечивает создание эффективного ключа, т.к. код в других типах может стать помехой вашим аннотациям. Более удачный подход предполагает использование внутреннего или (вложенного) закрытого класса:

```
class X
{
  class CustomData { internal string Message; }  // Закрытый вложенный тип

  static void Test()
  {
    XElement e = new XElement ("test");
    e.AddAnnotation (new CustomData { Message = "Hello" } );
    Console.Write (e.Annotations<CustomData>().First().Message);  // Hello
  }
}
```

Для удаления аннотаций вы должны иметь доступ также и к типу ключа:

```
e.RemoveAnnotations<CustomData>();
```

# Проецирование в модель X-DOM

До сих пор мы показывали, как применять LINQ для получения данных *из* модели X-DOM. Запросы LINQ можно также использовать для проецирования *в* модель X-DOM. Источником может быть все, к чему поддерживаются запросы LINQ, в том числе:

- запросы LINQ to SQL или Entity Framework;
- локальная коллекция;
- другая модель X-DOM.

Независимо от источника применяется та же самая стратегия, что и в случае использования LINQ для выдачи дерева X-DOM: сначала записывается выражение *функционального построения*, которое создает желаемую форму X-DOM, а затем на основе этого выражения строится запрос LINQ.

Например, пусть необходимо извлекать заказчиков из базы данных в XML-код следующего вида:

```
<customers>
  <customer id="1">
    <name>Sue</name>
    <buys>3</buys>
  </customer>
  ...
</customers>
```

Мы начинаем с того, что представляем выражение функционального построения для X-DOM, применяя простые литералы:

```
var customers =
  new XElement ("customers",
    new XElement ("customer", new XAttribute ("id", 1),
      new XElement ("name", "Sue"),
      new XElement ("buys", 3)
    )
  );
```

Затем мы превращаем это выражение в проекцию и строим на его основе запрос LINQ:

```
var customers =
  new XElement ("customers",
    from c in dataContext.Customers
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
```

В Entity Framework после извлечения заказчиков потребуется вызов `.ToList()`, а потому третья строка будет выглядеть так:

```
from c in objectContext.Customers.ToList()
```

Ниже показан результат:

```
<customers>
  <customer id="1">
    <name>Tom</name>
    <buys>3</buys>
  </customer>
  <customer id="2">
    <name>Harry</name>
    <buys>2</buys>
  </customer>
    ...
</customers>
```

Чтобы лучше понять, как все работает, сконструируем тот же самый запрос за два шага. Вот первый шаг:

```
IEnumerable<XElement> sqlQuery =
  from c in dataContext.Customers
  select
    new XElement ("customer", new XAttribute ("id", c.ID),
      new XElement ("name", c.Name),
      new XElement ("buys", c.Purchases.Count)
    );
```

Внутренняя порция представляет собой нормальный запрос LINQ to SQL, который выполняет проецирование в специальные типы (с точки зрения LINQ to SQL). Вот второй шаг:

```
var customers = new XElement ("customers", sqlQuery);
```

Здесь конструируется корневой элемент `XElement`. Единственный необычный аспект заключается в том, что содержимое, т.е. `sqlQuery` – это не одиночный `XElement`, а реализация `IQueryable<XElement>`, которая в свою очередь реализует интерфейс `IEnumerable<XElement>`. Вспомните, что при обработке XML-содержимого происходит автоматическое перечисление коллекций. Таким образом, каждый `XElement` добавляется как дочерний узел.

Внешний запрос также определяет линию, на которой запрос переходит от запроса базы данных через локальный LINQ в перечислимый запрос. Конструктору класса `XElement` ничего не известно об интерфейсе `IQueryable<>`, поэтому он принудительно инициирует перечисление запроса базы данных – и выполнение SQL-оператора.

## Устранение пустых элементов

Предположим, что в предыдущем примере также необходимо включить подробности о последней дорогой покупке заказчика. Решить задачу можно было бы следующим образом:

```
var customers =
  new XElement ("customers",
    from c in dataContext.Customers
    let lastBigBuy = (from p in c.Purchases
                      where p.Price > 1000
                      orderby p.Date descending
                      select p).FirstOrDefault()
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count),
```

```
      new XElement ("lastBigBuy",
        new XElement ("description", lastBigBuy?.Description,
        new XElement ("price", lastBigBuy?.Price ?? 0m)
      )
    )
);
```

Однако здесь будут выпускаться пустые элементы для заказчиков, не совершивших дорогих покупок. (Если бы это был локальный запрос, а не запрос к базе данных, то сгенерировалось бы исключение `NullReferenceException`.) В таких случаях лучше полностью опустить узел `lastBigBuy`, поместив конструктор для элемента `lastBigBuy` внутрь условной операции:

```
select
  new XElement ("customer", new XAttribute ("id", c.ID),
    new XElement ("name", c.Name),
    new XElement ("buys", c.Purchases.Count),
    lastBigBuy == null ? null :
      new XElement ("lastBigBuy",
        new XElement ("description", lastBigBuy.Description),
        new XElement ("price", lastBigBuy.Price)
```

Для заказчиков, не имеющих `lastBigBuy`, вместо пустого элемента `XElement` выдается значение `null`. Это именно то, что нужно, т.к. содержимое `null` попросту игнорируется.

## Потоковая передача проекции

Если проецирование в модель X-DOM осуществляется только с целью его сохранения посредством метода `Save` (или вызова `ToString`), то эффективность использования памяти можно повысить, задействовав класс `XStreamingElement`. Класс `XStreamingElement` представляет собой усеченную версию `XElement`, которая применяет семантику *отложенной загрузки* к своему дочернему содержимому. Для его использования нужно просто заменить внешние элементы `XElement` элементами `XStreamingElement`:

```
var customers =
  new XStreamingElement ("customers",
    from c in dataContext.Customers
    select
      new XStreamingElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
customers.Save ("data.xml");
```

Запросы, переданные конструктору `XStreamingElement`, не перечисляются вплоть до вызова метода `Save`, `ToString` или `WriteTo` на элементе, что позволяет избежать загрузки в память сразу целого дерева X-DOM. Оборотной стороной такого подхода является то, что запросы оцениваются повторно, требуя сохранения заново. Кроме того, обход дочернего содержимого `XStreamingElement` невозможен – данный класс не открывает доступ к методам вроде `Elements` или `Attributes`.

Класс `XStreamingElement` не основан на `XObject` (или на любом другом классе), поэтому он располагает таким ограниченным набором членов.

В состав членов помимо Save, ToString и WriteTo входят:

- метод Add, который принимает содержимое подобно конструктору;
- свойство Name.

Класс XStreamingElement не позволяет *читать* содержимое в потоковой манере – в таком случае придется применять класс XmlReader в сочетании с X-DOM. Мы объясним, как это делать, в разделе "Шаблоны для использования XmlReader/XmlWriter" главы 11.

## Трансформирование X-DOM

Модель X-DOM можно трансформировать путем ее повторного проецирования. Например, предположим, что необходимо трансформировать XML-файл *MSBuild*, используемый компилятором C# и средой Visual Studio для описания проекта, в простой формат, подходящий для генерации отчета. Содержимое файла MSBuild выглядит следующим образом:

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/dev...>
  <PropertyGroup>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>9.0.11209</ProductVersion>
    ...
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="ObjectGraph.cs" />
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
    <Compile Include="Tests\Aggregation.cs" />
    <Compile Include="Tests\Advanced\RecursiveXml.cs" />
  </ItemGroup>
  <ItemGroup>
    ...
  </ItemGroup>
  ...
</Project>
```

Пусть нам требуется включать только файлы, как показано ниже:

```
<ProjectReport>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <File>Properties\AssemblyInfo.cs</File>
  <File>Tests\Aggregation.cs</File>
  <File>Tests\Advanced\RecursiveXml.cs</File>
</ProjectReport>
```

Такую трансформацию выполняет следующий запрос:

```
XElement project = XElement.Load ("myProjectFile.csproj");
XNamespace ns = project.Name.Namespace;
var query =
  new XElement ("ProjectReport",
    from compileItem in
      project.Elements (ns + "ItemGroup").Elements (ns + "Compile")
    let include = compileItem.Attribute ("Include")
    where include != null
    select new XElement ("File", include.Value)
  );
```

Запрос сначала извлекает все элементы `ItemGroup`, а затем с помощью расширяющего метода `Elements` получает плоскую последовательность всех их подэлементов `Compile`. Обратите внимание, что мы указали пространство имен XML (все в исходном файле наследует пространство имен, определенное элементом `Project`), поэтому имя локального элемента, такое как `ItemGroup`, не будет работать само по себе. Затем извлекается значение атрибута `Include` и проецируется как элемент.

### Более сложные трансформации

При запросе локальной коллекции, подобной X-DOM, вы можете записывать специальные операции запросов для построения более сложных запросов.

Предположим, что в предыдущем примере необходимо получить иерархический вывод, основанный на папках:

```
<Project>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <Folder name="Properties">
    <File>AssemblyInfo.cs</File>
  </Folder>
  <Folder name="Tests">
    <File>Aggregation.cs</File>
    <Folder name="Advanced">
      <File>RecursiveXml.cs</File>
    </Folder>
  </Folder>
</Project>
```

Для построения такого вывода понадобится рекурсивно обрабатывать строки путей вроде *Tests\Advanced\RecursiveXml.cs*. Именно это делает показанный ниже метод: он принимает последовательность строк путей и выпускает иерархию X-DOM, соответствующую желаемому выводу:

```
static IEnumerable<XElement> ExpandPaths (IEnumerable<string> paths)
{
  var brokenUp = from path in paths
                 let split = path.Split (new char[] { '\\' }, 2)
                 orderby split[0]
                 select new
                 {
                   name = split[0],
                   remainder = split.ElementAtOrDefault (1)
                 };

  IEnumerable<XElement> files = from b in brokenUp
                                where b.remainder == null
                                select new XElement ("file", b.name);

  IEnumerable<XElement> folders = from b in brokenUp
                                  where b.remainder != null
                                  group b.remainder by b.name into grp
                                  select new XElement ("folder",
                                    new XAttribute ("name", grp.Key),
                                    ExpandPaths (grp)
                                  );
  return files.Concat (folders);
}
```

Первый запрос разбивает каждую строку пути по первой обратной косой черте на части name и remainder:

```
Tests\Advanced\RecursiveXml.cs -> Tests + Advanced\RecursiveXml.cs
```

Если значение remainder равно null, то мы имеем дело с простым именем файла. В таких случаях извлечением занимается запрос files.

Если значение remainder не равно null, тогда мы имеем дело с папкой. Такие случаи обрабатываются запросом folders. Поскольку в папке могут находиться и другие файлы, необходимо сгруппировать их вместе по имени папки с помощью group. Для каждой группы затем выполняется одна и та же функция на ее элементах.

Окончательным результатом будет конкатенация files и folders. Операция Concat сохраняет порядок, потому сначала в алфавитном порядке идут файлы, а за ними в алфавитном порядке следуют папки.

Имея данный метод, мы можем построить запрос за два шага. Первым делом мы извлекаем простую последовательность строк путей:

```
IEnumerable<string> paths =
  from compileItem in
    project.Elements (ns + "ItemGroup").Elements (ns + "Compile")
  let include = compileItem.Attribute ("Include")
  where include != null
  select include.Value;
```

Затем мы передаем эту последовательность методу ExpandPaths для получения финального результата:

```
var query = new XElement ("Project", ExpandPaths (paths));
```

# 11

# Другие технологии XML

Пространство имен `System.Xml` содержит следующие пространства имен и основные классы.

**`XmlReader` и `XmlWriter`**

Высокопроизводительные однонаправленные курсоры для чтения и записи в XML-поток.

**`XmlDocument`**

Представление XML-документа в DOM-модели стиля W3C (устарел).

**`System.Xml.XLinq`**

Современная DOM-модель, ориентированная на LINQ, для работы с XML.

**`System.Xml.XmlSchema`**

Инфраструктура и API-интерфейс для схем XSD (W3C).

**`System.Xml.Xsl`**

Инфраструктура и API-интерфейс (`XslCompiledTransform`) для выполнения трансформаций XSLT структур XML (W3C).

**`System.Xml.Serialization`**

Поддержка сериализации классов в и из XML-данных (глава 17).

Здесь W3C – это аббревиатура, обозначающая консорциум World Wide Web Consortium, который определяет стандарты XML.

Статический класс `XmlConvert`, предназначенный для разбора и форматирования XML-строк, был описан в главе 6.

## `XmlReader`

`XmlReader` – высокопроизводительный класс для чтения XML-потока низкоуровневым однонаправленным способом.

Рассмотрим следующее содержимое XML-файла:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Для создания экземпляра `XmlReader` вызывается статический метод `XmlReader.Create`, которому передается объект `Stream`, объект `TextReader` или строка URI. Например:

```
using (XmlReader reader = XmlReader.Create ("customer.xml"))
  ...
```

Поскольку класс `XmlReader` позволяет читать из потенциально медленных источников (`Stream` и URI), он предлагает асинхронные версии большинства своих методов, так что можно легко писать неблокирующий код. Асинхронность подробно обсуждается в главе 14.

Ниже показано, как сконструировать экземпляр `XmlReader`, который читает из строки:

```
XmlReader reader = XmlReader.Create (
  new System.IO.StringReader (myString));
```

Для управления настройками разбора и проверки достоверности можно также передавать объект `XmlReaderSettings`. В частности, следующие три свойства `XmlReaderSettings` полезны при пропускании избыточного содержимого:

```
bool IgnoreComments                // Пропускать узлы комментариев?
bool IgnoreProcessingInstructions // Пропускать инструкции обработки?
bool IgnoreWhitespace              // Пропускать пробельные символы?
```

В приведенном далее примере средству чтения сообщается о том, что узлы с пробельными символами, которые отвлекают внимание в типовых сценариях, выпускаться не должны:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
  ...
```

Еще одним полезным свойством `XmlReaderSettings` является `ConformanceLevel`. Его стандартное значение `Document` указывает средству чтения на то, что необходимо предполагать наличие допустимого XML-документа с единственным корневым узлом. Такая проблема возникает, когда нужно прочитать только внутреннюю порцию XML-кода, содержащую несколько узлов:

```
<firstname>Jim</firstname>
<lastname>Bo</lastname>
```

Чтобы прочитать такой XML-код без генерации исключения, потребуется установить `ConformanceLevel` в `Fragment`.

Класс `XmlReaderSettings` также имеет свойство по имени `CloseInput`, которое указывает на то, должен ли закрываться лежащий в основе поток, когда закрывается средство чтения (в `XmlWriterSettings` существует аналогичное свойство под названием `CloseOutput`). Стандартное значение для свойств `CloseInput` и `CloseOutput` равно `false`.

## Чтение узлов

Единицами XML-потока являются *узлы XML*. Средство чтения перемещается по потоку в текстовом порядке (сначала в глубину). Свойство `Depth` средства чтения возвращает текущую глубину курсора.

Самый простой способ чтения из `XmlReader` предполагает вызов метода `Read`. Он осуществляет перемещение на следующий узел в XML-потоке подобно методу `MoveNext` из интерфейса `IEnumerator`. Первый вызов `Read` устанавливает курсор на первый узел. Когда метод `Read` возвращает `false`, это означает, что курсор переместился *за* последний узел, и в данном случае экземпляр `XmlReader` должен быть закрыт и освобожден.

В следующем примере мы читаем каждый узел в XML-потоке, выводя тип узла по мере продвижения:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
  while (reader.Read())
  {
    Console.Write (new string (' ',reader.Depth*2));  // Вывести отступ
    Console.WriteLine (reader.NodeType);
  }
```

Ниже показан вывод:

```
XmlDeclaration
Element
  Element
    Text
  EndElement
  Element
    Text
  EndElement
EndElement
```

Атрибуты в обход на основе `Read` не включаются (см. раздел "Чтение атрибутов" далее в главе).

Свойство `NodeType` имеет тип `XmlNodeType`, который представляет собой перечисление со следующими членами:

| | | |
|---|---|---|
| `None` | `Comment` | `Document` |
| `XmlDeclaration` | `Entity` | `DocumentType` |
| `Element` | `EndEntity` | `DocumentFragment` |
| `EndElement` | `EntityReference` | `Notation` |
| `Text` | `ProcessingInstruction` | `Whitespace` |
| `Attribute` | `CDATA` | `SignificantWhitespace` |

Два строковых свойства в `XmlReader` — `Name` и `Value` — обеспечивают доступ к содержимому узла. В зависимости от типа узла, наполняется либо `Name`, либо `Value` (или оба):

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
settings.DtdProcessing = DtdProcessing.Parse;  // Требуется для чтения DTD
```

```
using (XmlReader r = XmlReader.Create ("customer.xml", settings))
  while (r.Read())
  {
    Console.Write (r.NodeType.ToString().PadRight (17, '-'));
    Console.Write ("> ".PadRight (r.Depth * 3));

    switch (r.NodeType)
    {
      case XmlNodeType.Element:
      case XmlNodeType.EndElement:
        Console.WriteLine (r.Name); break;

      case XmlNodeType.Text:
      case XmlNodeType.CDATA:
      case XmlNodeType.Comment:
      case XmlNodeType.XmlDeclaration:
        Console.WriteLine (r.Value); break;

      case XmlNodeType.DocumentType:
        Console.WriteLine (r.Name + " - " + r.Value); break;

      default: break;
    }
  }
```

В целях демонстрации мы расширим наш XML-файл, включив тип документа, сущность, CDATA и комментарий:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE customer [ <!ENTITY tc "Top Customer"> ]>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <quote><![CDATA[C#'s operators include: < > &]]></quote>
  <notes>Jim Bo is a &tc;</notes>
  <!--  That wasn't so bad! -->
</customer>
```

Сущность подобна макросу, а CDATA похоже на дословную строку (`@"..."`) в C#. Ниже показан результат:

```
XmlDeclaration---> version="1.0" encoding="utf-8"
DocumentType-----> customer -  <!ENTITY tc "Top Customer">
Element----------> customer
Element---------->  firstname
Text------------->    Jim
EndElement------->  firstname
Element---------->  lastname
Text------------->    Bo
EndElement------->  lastname
Element---------->  quote
CDATA------------>    C#'s operators include: < > &
EndElement------->  quote
Element---------->  notes
Text------------->    Jim Bo is a Top Customer
EndElement------->  notes
Comment---------->   That wasn't so bad!
EndElement------->  customer
```

Класс `XmlReader` автоматически распознает сущности, так что в рассмотренном примере ссылка на сущность `&tc;` расширяется в `Top Customer`.

## Чтение элементов

Зачастую структура читаемого XML-документа уже известна. Чтобы помочь в этом отношении, класс `XmlReader` предлагает набор методов, которые выполняют чтение, *предполагая* наличие определенной структуры. Они упрощают код и одновременно предпринимают некоторую проверку достоверности.

Класс `XmlReader` генерирует исключение `XmlException`, если любая проверка достоверности терпит неудачу. Класс `XmlException` имеет свойства `LineNumber` и `LinePosition`, которые указывают, где произошла ошибка – в случае крупных XML-файлов регистрация такой информации в журнале очень важна!

Метод `ReadStartElement` проверяет, что текущий `NodeType` является `Element`, и затем вызывает метод `Read`. Если указано имя, тогда он проверяет, что оно совпадает с именем текущего элемента. Метод `ReadEndElement` удостоверяется в том, что текущий `NodeType` – это `EndElement`, и затем вызывает метод `Read`.

Например, мы могли бы прочитать узел:

```
<firstname>Jim</firstname>
```

следующим образом:

```
reader.ReadStartElement ("firstname");
Console.WriteLine (reader.Value);
reader.Read();
reader.ReadEndElement();
```

Метод `ReadElementContentAsString` выполняет все описанные ранее действия за раз. Он читает начальный элемент, текстовый узел и конечный элемент, возвращая содержимое в виде строки:

```
string firstName = reader.ReadElementContentAsString ("firstname", "");
```

Второй аргумент ссылается на пространство имен, которое в приведенном примере оставлено пустым. Доступны также типизированные версии метода, такие как `ReadElementContentAsInt`, разбирающие результат. Вернемся к исходному XML-документу:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <creditlimit>500.00</creditlimit> <!--Да, мы не учитываем этот комментарий!-->
</customer>
```

Его можно прочитать следующим образом:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
using (XmlReader r = XmlReader.Create ("customer.xml", settings))
{
  r.MoveToContent();   // Пропустить XML-объявление
  r.ReadStartElement ("customer");
  string firstName    = r.ReadElementContentAsString ("firstname", "");
  string lastName     = r.ReadElementContentAsString ("lastname", "");
  decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");
  r.MoveToContent();   // Пропустить этот надоедливый комментарий
  r.ReadEndElement();  // Прочитать закрывающий дескриптор customer
}
```

Метод MoveToContent по-настоящему удобен. Он пропускает все малоинтересное: XML-объявления, пробельные символы, комментарии и инструкции обработки. Посредством свойств XmlReaderSettings можно заставить средство чтения выполнять большинство таких действий автоматически.

### Необязательные элементы

Предположим, что в предыдущем примере элемент <lastname> был необязательным. Решение прямолинейно:

```
r.ReadStartElement ("customer");
string firstName    = r. ReadElementContentAsString ("firstname", "");
string lastName     = r.Name == "lastname"
                    ? r.ReadElementContentAsString() : null;
decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");
```

### Случайный порядок элементов

Примеры в текущем разделе полагаются на то, что элементы в XML-файле расположены в установленном порядке. Чтобы справиться с элементами, представленными в другом порядке, проще всего прочитать такой раздел XML-файла в дерево X-DOM. Мы покажем, как это делать, в разделе "Шаблоны для использования XmlReader/XmlWriter" далее в главе.

### Пустые элементы

Способ, которым класс XmlReader обрабатывает пустые элементы, таит в себе серьезную ловушку. Рассмотрим следующий элемент:

```
<customerList></customerList>
```

Вот его эквивалент в XML:

```
<customerList/>
```

Тем не менее, XmlReader трактует два варианта по-разному. В первом случае приведенный ниже код работает ожидаемым образом:

```
reader.ReadStartElement ("customerList");
reader.ReadEndElement();
```

Во втором случае метод ReadEndElement генерирует исключение, т.к. отсутствует отдельный "конечный элемент", на который рассчитывает класс XmlReader. Обходной путь предусматривает добавление проверки на предмет пустых элементов:

```
bool isEmpty = reader.IsEmptyElement;
reader.ReadStartElement ("customerList");
if (!isEmpty) reader.ReadEndElement();
```

На самом деле такая неприятность возникает, только когда рассматриваемый элемент может содержать дочерние элементы (скажем, список заказчиков). В случае элементов, которые содержат простой текст (вроде firstname), проблемы можно избежать путем вызова такого метода, как ReadElementContentAsString. Методы ReadElement*XXX* корректно обрабатывают оба вида пустых элементов.

### Другие методы Read*XXX*

В табл. 11.1 приведена сводка по всем методам Read*XXX* в классе XmlReader. Большинство из них предназначено для работы с элементами. Выделенная полужирным часть в примере XML-фрагмента – это раздел, который читает описываемый метод.

**Таблица 11.1. Методы чтения**

| Методы | Типы узлов, на которых методы работают | Пример XML-фрагмента | Входные параметры | Возвращаемые данные |
|---|---|---|---|---|
| `ReadContentAs`*XXX* | `Text` | `<a>`**`x`**`</a>` | | `x` |
| `ReadString` | `Text` | `<a>`**`x`**`</a>` | | `x` |
| `ReadElementString` | `Element` | `<a>x</a>` | | `x` |
| `ReadElementContentAs`*XXX* | `Element` | `<a>x</a>` | | `x` |
| `ReadInnerXml` | `Element` | `<a>x</a>` | | `x` |
| `ReadOuterXml` | `Element` | `<a>x</a>` | | `<a>x</a>` |
| `ReadStartElement` | `Element` | **`<a>`**`x</a>` | | |
| `ReadEndElement` | `Element` | `<a>x`**`</a>`** | | |
| `ReadSubtree` | `Element` | `<a>x</a>` | | `<a>x</a>` |
| `ReadToDescendant` | `Element` | **`<a>x`**`<b></b></a>` | `"b"` | |
| `ReadToFollowing` | `Element` | **`<a>x`**`<b></b></a>` | `"b"` | |
| `ReadToNextSibling` | `Element` | **`<a>x</a>`**`<b></b>` | `"b"` | |
| `ReadAttributeValue` | `Attribute` | См. раздел "Чтение атрибутов" далее в главе | | |

Методы `ReadContentAs`*XXX* разбирают текстовый узел в тип *XXX*. Внутренне класс `XmlConvert` выполняет преобразование из строки в данный тип. Текстовый узел может находиться внутри элемента или атрибута.

Методы `ReadElementContentAs`*XXX* представляют собой оболочки вокруг соответствующих методов `ReadContentAs`*XXX*. Они применяются к узлу *элемента*, а не к *текстовому* узлу, заключенному в элемент.

Типизированные методы `Read`*XXX* также имеют версии, которые читают в байтовый массив данные в форматах Base 64 и BinHex.

Метод `ReadInnerXml` обычно применяется к элементу; он читает и возвращает элемент со всеми его потомками. В случае применения к атрибуту метод `ReadInnerXml` возвращает значение атрибута.

Метод `ReadOuterXml` аналогичен `ReadInnerXml`, но только включает, а не исключает элемент в позиции курсора.

Метод `ReadSubtree` возвращает новый экземпляр `XmlReader`, который обеспечивает представление только текущего элемента (и его потомков). Чтобы исходный `XmlReader` мог безопасно продолжить чтение, этот экземпляр должен быть закрыт. В момент, когда новый экземпляр `XmlReader` закрывается, позиция курсора исходного `XmlReader` перемещается в конец поддерева.

Метод `ReadToDescendant` перемещает курсор в начало первого узла-потомка с указанным именем/пространством имен.

Метод `ReadToFollowing` перемещает курсор в начало первого узла – независимо от глубины – с указанным именем/пространством имен.

Метод `ReadToNextSibling` перемещает курсор в начало первого родственного узла с указанным именем/пространством имен.

Методы `ReadString` и `ReadElementString` ведут себя подобно методам `ReadContentAsString` и `ReadElementContentAsString`, но с тем отличием, что генерируют исключение, если внутри элемента обнаруживается более *одного* текстового узла. В общем случае использования упомянутых методов следует избегать, потому что они генерируют исключение, если элемент содержит комментарий.

## Чтение атрибутов

Класс `XmlReader` предоставляет индексатор, обеспечивающий прямой (произвольный) доступ к атрибутам элемента – по имени или по позиции. Применение индексатора эквивалентно вызову метода `GetAttribute`.

Имея следующий XML-фрагмент:

```
<customer id="123" status="archived"/>
```

мы могли бы прочитать его атрибуты так:

```
Console.WriteLine (reader ["id"]);              // 123
Console.WriteLine (reader ["status"]);          // archived
Console.WriteLine (reader ["bogus"] == null);   // True
```

Для того чтобы читать атрибуты, экземпляр `XmlReader` должен располагаться *на начальном элементе. После* вызова метода `ReadStartElement` атрибуты исчезают навсегда!

Хотя порядок атрибутов семантически несущественен, доступ к атрибутам возможен по их ординальным позициям. Предыдущий пример можно переписать следующим образом:

```
Console.WriteLine (reader [0]);  // 123
Console.WriteLine (reader [1]);  // archived
```

Индексатор также позволяет указывать пространство имен атрибута, если оно имеется. Свойство `AttributeCount` возвращает количество атрибутов для текущего узла.

### Узлы атрибутов

Для явного обхода узлов атрибутов потребуется сделать специальное отклонение от нормального пути, предусматривающего просто вызов метода `Read`. Веской причиной поступить так является необходимость разбора значений атрибутов в другие типы с помощью методов `ReadContentAs`*XXX*.

Отклонение должно начинаться с *начального элемента*. В целях упрощения работы во время обхода атрибутов правило однонаправленности ослабляется: можно переходить к любому атрибуту (вперед или назад) за счет вызова метода `MoveToAttribute`.

Метод `MoveToElement` возвращает начальный элемент из любого места внутри ответвления узла атрибута.

Вернувшись к предыдущему примеру:

```
<customer id="123" status="archived"/>
```

можно поступить так:

```
reader.MoveToAttribute ("status");
string status = reader.ReadContentAsString();
reader.MoveToAttribute ("id");
int id = reader.ReadContentAsInt();
```

Метод MoveToAttribute возвращает false, если указанный атрибут не существует.

Можно также совершить обход всех атрибутов в последовательности, вызывая метод MoveToFirstAttribute, а затем метод MoveToNextAttribute:

```
if (reader.MoveToFirstAttribute())
  do
  {
    Console.WriteLine (reader.Name + "=" + reader.Value);
  }
  while (reader.MoveToNextAttribute());
// ВЫВОД:
id=123
status=archived
```

## Пространства имен и префиксы

Класс XmlReader предлагает две параллельные системы для ссылки на имена элементов и атрибутов:

- Name
- NamespaceURI и LocalName

Всякий раз, когда читается свойство Name элемента или вызывается метод, принимающий одиночный аргумент name, используется первая система. Такой подход хорошо работает в отсутствие каких-либо пространств имен или префиксов; в противном случае он действует в грубой и буквальной манере. Пространства имен игнорируются, а префиксы включаются в точности так, как они записаны. Например:

| Пример фрагмента | Значение Name |
|---|---|
| <customer ...> | customer |
| <customer xmlns='blah' ...> | customer |
| <x:customer ...> | x:customer |

Приведенный ниже код работает с первыми двумя случаями:

```
reader.ReadStartElement ("customer");
```

Для обработки третьего случая требуется следующий код:

```
reader.ReadStartElement ("x:customer");
```

Вторая система работает через два свойства, *осведомленные о пространствах имен*: NamespaceURI и LocalName. Указанные свойства принимают во внимание префиксы и стандартные пространства имен, определенные родительскими элементами. Префиксы автоматически расширяются. Это означает, что свойство NamespaceURI всегда отражает семантически корректное пространство имен для текущего элемента, а свойство LocalName всегда свободно от префиксов.

При передаче двух аргументов имен в такой метод, как ReadStartElement, вы применяете ту же самую систему. Например, взгляните на следующий XML-фрагмент:

```
<customer xmlns="DefaultNamespace" xmlns:other="OtherNamespace">
  <address>
    <other:city>
    ...
```

Прочитать его можно было бы так:

```
reader.ReadStartElement ("customer", "DefaultNamespace");
reader.ReadStartElement ("address",  "DefaultNamespace");
reader.ReadStartElement ("city",     "OtherNamespace");
```

Абстрагирование от префиксов обычно является именно тем, что нужно. При необходимости посредством свойства `Prefix` можно просмотреть, какой префикс использовался, и с помощью метода `LookupNamespace` преобразовать его в пространство имен.

# XmlWriter

Класс `XmlWriter` — это однонаправленное средство записи в XML-поток. Проектное решение, положенное в основу `XmlWriter`, симметрично таковому в классе `XmlReader`.

Как и `XmlTextReader`, экземпляр `XmlWriter` конструируется вызовом метода `Create`, которому передается необязательный объект настроек. В приведенном ниже примере мы разрешаем отступы, чтобы сделать вывод удобным для восприятия человеком, и затем записываем его в простой XML-файл:

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;

using (XmlWriter writer = XmlWriter.Create ("..\\..\\foo.xml", settings))
{
  writer.WriteStartElement ("customer");
  writer.WriteElementString ("firstname", "Jim");
  writer.WriteElementString ("lastname"," Bo");
  writer.WriteEndElement();
}
```

В результате получается следующий документ (тот же самый, что и в файле, который мы читали в первом примере применения класса `XmlReader`):

```
<?xml version="1.0" encoding="utf-8" ?>
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Класс `XmlWriter` автоматически записывает объявление в начале, если только в `XmlWriterSettings` не указано обратное за счет установки свойства `OmitXmlDeclaration` в `true` или свойства `ConformanceLevel` в `Fragment`. В последнем случае также разрешена запись нескольких корневых узлов — то, что иначе приводит к генерации исключения.

Метод `WriteValue` записывает одиночный текстовый узел. Он принимает строковые и нестроковые типы, такие как `bool` и `DateTime`, внутренне используя класс `XmlConvert` для выполнения совместимых с XML преобразований строк:

```
writer.WriteStartElement ("birthdate");
writer.WriteValue (DateTime.Now);
writer.WriteEndElement();
```

Напротив, если мы вызовем:

```
WriteElementString ("birthdate", DateTime.Now.ToString());
```

то результат окажется несовместимым с XML и уязвимым к некорректному разбору.

Вызов метода `WriteString` эквивалентен вызову метода `WriteValue` со строкой. Класс `XmlWriter` автоматически защищает символы, которые в противном случае были бы недопустимыми внутри атрибута либо элемента, такие как &, <, >, и расширенные символы Unicode.

## Запись атрибутов

Атрибуты можно записывать немедленно после записи начального элемента:

```
writer.WriteStartElement ("customer");
writer.WriteAttributeString ("id", "1");
writer.WriteAttributeString ("status", "archived");
```

Для записи нестроковых значений нужно вызывать методы `WriteStartAttribute`, `WriteValue` и `WriteEndAttribute`.

## Запись других типов узлов

В классе `XmlWriter` также определены следующие методы для записи других разновидностей узлов:

```
WriteBase64  // для двоичных данных
WriteBinHex  // для двоичных данных
WriteCData
WriteComment
WriteDocType
WriteEntityRef
WriteProcessingInstruction
WriteRaw
WriteWhitespace
```

Метод `WriteRaw` внедряет строку прямо в выходной поток. Имеется также метод `WriteNode`, который принимает экземпляр `XmlReader` и копирует из него все данные.

## Пространства имен и префиксы

Перегруженные версии методов `Write*` позволяют ассоциировать элемент или атрибут с пространством имен. Давайте перепишем содержимое XML-файла из предыдущего примера. На этот раз мы будем связывать все элементы с пространством имен `http://oreilly.com`, объявив префикс `o` в элементе `customer`:

```
writer.WriteStartElement ("o", "customer", "http://oreilly.com");
writer.WriteElementString ("o", "firstname", "http://oreilly.com", "Jim");
writer.WriteElementString ("o", "lastname", "http://oreilly.com", "Bo");
writer.WriteEndElement();
```

Вывод теперь выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<o:customer xmlns:o='http://oreilly.com'>
  <o:firstname>Jim</o:firstname>
  <o:lastname>Bo</o:lastname>
</o:customer>
```

Обратите внимание, что для краткости класс XmlWriter опускает объявления пространств имен в дочерних элементах, если они уже объявлены их родительским элементом.

# Шаблоны для использования XmlReader/XmlWriter

## Работа с иерархическими данными

Рассмотрим следующие классы:

```
public class Contacts
{
  public IList<Customer> Customers = new List<Customer>();
  public IList<Supplier> Suppliers = new List<Supplier>();
}

public class Customer { public string FirstName, LastName; }
public class Supplier { public string Name;                 }
```

Предположим, что мы хотим применить классы XmlReader и XmlWriter для сериализации объекта Contacts в XML, как в приведенном фрагменте:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<contacts>
  <customer id="1">
    <firstname>Jay</firstname>
    <lastname>Dee</lastname>
  </customer>
  <customer>  <!-- мы будем предполагать, что id необязателен -->
    <firstname>Kay</firstname>
    <lastname>Gee</lastname>
  </customer>
  <supplier>
    <name>X Technologies Ltd</name>
  </supplier>
</contacts>
```

Лучший подход заключается в том, чтобы не записывать один крупный метод, а инкапсулировать XML-функциональность в самих типах Customer и Supplier, реализовав для них методы ReadXml и WriteXml. Используемый шаблон довольно прост:

- когда методы ReadXml и WriteXml завершаются, они оставляют средство чтения/записи на той же глубине;
- метод ReadXml читает внешний элемент, тогда как метод WriteXml записывает только его внутреннее содержимое.

Ниже показано, как можно было бы реализовать тип Customer:

```
public class Customer
{
  public const string XmlName = "customer";
  public int? ID;
  public string FirstName, LastName;

  public Customer () { }
  public Customer (XmlReader r) { ReadXml (r); }
```

```
  public void ReadXml (XmlReader r)
  {
    if (r.MoveToAttribute ("id")) ID = r.ReadContentAsInt();
    r.ReadStartElement();
    FirstName = r.ReadElementContentAsString ("firstname", "");
    LastName = r.ReadElementContentAsString ("lastname", "");
    r.ReadEndElement();
  }
  public void WriteXml (XmlWriter w)
  {
    if (ID.HasValue) w.WriteAttributeString ("id", "", ID.ToString());
    w.WriteElementString ("firstname", FirstName);
    w.WriteElementString ("lastname", LastName);
  }
}
```

Обратите внимание, что метод `ReadXml` читает узлы внешнего начального и конечного элементов. Если бы эту работу делал вызывающий компонент, то класс `Customer` мог бы не читать собственные атрибуты. Причина, по которой метод `WriteXml` не сделан симметричным в таком отношении, двойственна:

- вызывающий компонент может нуждаться в выборе способа именования внешнего элемента;
- вызывающему компоненту может быть необходима запись дополнительных XML-атрибутов, таких как *подтип* элемента (который затем может применяться для принятия решения о том, экземпляр какого класса создавать при чтении данного элемента).

Еще одно преимущество следования описанному шаблону связано с тем, что ваша реализация будет совместимой с интерфейсом `IXmlSerializable` (глава 17).

Класс `Supplier` аналогичен:

```
public class Supplier
{
  public const string XmlName = "supplier";
  public string Name;

  public Supplier () { }
  public Supplier (XmlReader r) { ReadXml (r); }

  public void ReadXml (XmlReader r)
  {
    r.ReadStartElement();
    Name = r.ReadElementContentAsString ("name", "");
    r.ReadEndElement();
  }

  public void WriteXml (XmlWriter w)
  {
    w.WriteElementString ("name", Name);
  }
}
```

В классе `Contacts` мы должны выполнять перечисление элемента `customers` в методе `ReadXml` с целью проверки, является ли каждый подэлемент заказчиком или поставщиком. Также понадобится закодировать обработку пустых элементов:

```
public void ReadXml (XmlReader r)
{
  bool isEmpty = r.IsEmptyElement; // Это обеспечивает корректную
  r.ReadStartElement();            // обработку пустого
  if (isEmpty) return;             // элемента <contacts/>
  while (r.NodeType == XmlNodeType.Element)
  {
    if (r.Name == Customer.XmlName)      Customers.Add (new Customer (r));
    else if (r.Name == Supplier.XmlName) Suppliers.Add (new Supplier (r));
    else
      throw new XmlException ("Unexpected node: " + r.Name);
                           // Непредвиденный узел
  }
  r.ReadEndElement();
}
public void WriteXml (XmlWriter w)
{
  foreach (Customer c in Customers)
  {
    w.WriteStartElement (Customer.XmlName);
    c.WriteXml (w);
    w.WriteEndElement();
  }
  foreach (Supplier s in Suppliers)
  {
    w.WriteStartElement (Supplier.XmlName);
    s.WriteXml (w);
    w.WriteEndElement();
  }
}
```

## Смешивание XmlReader/XmlWriter с моделью X-DOM

Переключиться на модель X-DOM можно в любой точке XML-дерева, где работа с классами `XmlReader` или `XmlWriter` становится слишком громоздкой. Использование X-DOM для обработки внутренних элементов – великолепный способ комбинирования простоты применения X-DOM и низкого расхода памяти классами `XmlReader` и `XmlWriter`.

### Использование XmlReader с XElement

Чтобы прочитать текущий элемент в модель X-DOM, необходимо вызвать метод `XNode.ReadFrom`, передав ему экземпляр `XmlReader`. В отличие от `XElement.Load` этот метод не является "жадным" в том смысле, что он не ожидает увидеть целый документ. Взамен метод `XNode.ReadFrom` читает только до конца текущего поддерева.

В качестве примера предположим, что имеется XML-файл журнала со следующей структурой:

```
<log>
  <logentry id="1">
    <date>...</date>
    <source>...</source>
    ...
  </logentry>
  ...
</log>
```

При наличии миллиона элементов logentry чтение целого журнала в модель X-DOM приведет к непроизводительному расходу памяти. Более эффективное решение предусматривает обход всех элементов logentry с помощью класса XmlReader и затем использование XElement для индивидуальной обработки каждого элемента:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
using (XmlReader r = XmlReader.Create ("logfile.xml", settings))
{
  r.ReadStartElement ("log");
  while (r.Name == "logentry")
  {
    XElement logEntry = (XElement) XNode.ReadFrom (r);
    int id = (int) logEntry.Attribute ("id");
    DateTime date = (DateTime) logEntry.Element ("date");
    string source = (string) logEntry.Element ("source");
    ...
  }
  r.ReadEndElement();
}
```

Если следовать шаблону, описанному в предыдущем разделе, тогда можно поместить XElement внутрь метода ReadXml или WriteXml специального типа так, что вызывающий компонент даже не обнаружит подвоха! Например, метод ReadXml класса Customer можно было бы переписать следующим образом:

```
public void ReadXml (XmlReader r)
{
  XElement x = (XElement) XNode.ReadFrom (r);
  FirstName = (string) x.Element ("firstname");
  LastName = (string) x.Element ("lastname");
}
```

Класс XElement взаимодействует с классом XmlReader, чтобы гарантировать, что пространства имен остались незатронутыми, а префиксы соответствующим образом расширенными – даже если они определены на внешнем уровне. Таким образом, если содержимое XML-файла выглядит, как показано ниже:

```
<log xmlns="http://loggingspace">
  <logentry id="1">
  ...
```

то экземпляры XElement, сконструированные на уровне logentry, будут корректно наследовать внешнее пространство имен.

### Использование XmlWriter с XElement

Класс XElement можно применять только для записи внутренних элементов в XmlWriter. В приведенном далее коде производится запись миллиона элементов logentry в XML-файл с использованием класса XElement – без помещения всех их в память:

```
using (XmlWriter w = XmlWriter.Create ("log.xml"))
{
  w.WriteStartElement ("log");
  for (int i = 0; i < 1000000; i++)
  {
    XElement e = new XElement ("logentry",
                 new XAttribute ("id", i),
```

```
                new XElement ("date", DateTime.Today.AddDays (-1)),
                new XElement ("source", "test"));
    e.WriteTo (w);
  }
  w.WriteEndElement ();
}
```

С применением класса `XElement` связаны минимальные накладные расходы во время выполнения. Если мы изменим пример для повсеместного использования класса `XmlWriter`, то никакой заметной разницы в скорости выполнения не будет.

# XSD и проверка достоверности с помощью схемы

Содержимое отдельного XML-документа почти всегда является специфичным для предметной области, как в случае документа Microsoft Word, документа с конфигурацией приложения или веб-службы. В каждой предметной области XML-файл соответствует определенному шаблону. Описание схем таких шаблонов регламентируется несколькими стандартами, которые предназначены для унификации и автоматизации процедур интерпретации и проверки достоверности XML-документов. Самым широко принятым стандартом является *XSD* (*XML Schema Definition* – определение схемы XML). Его предшественники, DTD и XDR, также поддерживаются пространством имен `System.Xml`.

Взгляните на следующий XML-документ:

```
<?xml version="1.0"?>
<customers>
  <customer id="1" status="active">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </customer>
  <customer id="1" status="archived">
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </customer>
</customers>
```

Определение XSD для этого документа можно записать так:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="customers">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="customer">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="firstname" type="xs:string" />
              <xs:element name="lastname" type="xs:string" />
            </xs:sequence>
            <xs:attribute name="id" type="xs:int" use="required" />
            <xs:attribute name="status" type="xs:string" use="required" />
          </xs:complexType>
```

```
      </xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Как видите, сами XSD-документы представляются с помощью XML. Более того, XSD-документ может быть описан посредством XSD – вы найдете такое определение по адресу `http://www.w3.org/2001/xmlschema.xsd`.

## Выполнение проверки достоверности с помощью схемы

Перед чтением или обработкой файл либо документ XML можно проверить на соответствие одной или нескольким схемам. Это делается по следующим причинам:

- можно уменьшить объем проверки на предмет ошибок и обработки исключений;
- проверка достоверности с помощью схемы позволяет обнаружить ошибки, которые в противном случае остались бы незамеченными;
- сообщения об ошибках являются подробными и информативными.

Для выполнения проверки достоверности необходимо подключить схему к объекту `XmlReader`, `XmlDocument` или X-DOM и затем читать либо загружать XML-данные обычным образом. Проверка достоверности посредством схемы происходит автоматически по мере чтения содержимого, так что входной поток не читается дважды.

### Проверка достоверности `XmlReader`

Ниже показано, как подключить схему из файла *`customers.xsd`* к объекту `XmlReader`:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");

using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  ...
```

Если схема является встроенной, тогда вместо добавления к свойству `Schemas` понадобится установить следующий флаг:

```
settings.ValidationFlags |= XmlSchemaValidationFlags.ProcessInlineSchema;
```

Затем можно выполнять чтение обычным образом. Если в какой-то момент происходит отказ при проверке достоверности посредством схемы, то генерируется исключение `XmlSchemaValidationException`.

Вызов метода `Read` сам по себе обеспечивает проверку достоверности и элементов, и атрибутов: переходить к каждому отдельному атрибуту с целью его проверки не придется.

Если требуется *только* проверить документ, тогда можно поступить так:

```
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { while (r.Read()) ; }
  catch (XmlSchemaValidationException ex)
  {
    ...
  }
```

Класс XmlSchemaValidationException имеет свойства Message, LineNumber и LinePosition. В данном случае он сообщает лишь о первой ошибке, обнаруженной в документе. Чтобы получить сведения обо всех ошибках в документе, потребуется организовать обработку события ValidationEventHandler:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");
settings.ValidationEventHandler += ValidationHandler;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  while (r.Read()) ;
```

Когда это событие обрабатывается, то ошибки, связанные со схемой, больше не приводят к генерации исключения. Взамен они запускают обработчик события:

```
static void ValidationHandler (object sender, ValidationEventArgs e)
{
  Console.WriteLine ("Error: " + e.Exception.Message); // сообщение об ошибке
}
```

Свойство Exception класса ValidationEventArgs содержит экземпляр исключения XmlSchemaValidationException, которое сгенерировалось бы в противном случае.

В пространстве имен System.Xml также определен класс по имени XmlValidatingReader. Он предназначен для выполнения проверки достоверности с помощью схемы в версиях, предшествующих .NET Framework 2.0, и в настоящее время считается устаревшим.

## Проверка достоверности X-DOM

Для выполнения проверки достоверности файла или потока XML во время его чтения в модель X-DOM необходимо создать экземпляр XmlReader, подключить схемы и применить средство чтения для загрузки DOM-модели:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");

XDocument doc;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { doc = XDocument.Load (r); }
  catch (XmlSchemaValidationException ex) { ... }
```

С помощью расширяющих методов из пространства имен System.Xml.Schema можно выполнять проверку достоверности объекта XDocument или XElement, уже находящегося в памяти. Эти методы принимают экземпляр XmlSchemaSet (коллекция схем) и обработчик событий проверки:

```
XDocument doc = XDocument.Load (@"customers.xml");
XmlSchemaSet set = new XmlSchemaSet ();
set.Add (null, @"customers.xsd");
StringBuilder errors = new StringBuilder ();
doc.Validate (set, (sender, args) => { errors.AppendLine
                                       (args.Exception.Message); }
              );
Console.WriteLine (errors.ToString());
```

# XSLT

Аббревиатура XSLT означает *Extensible Stylesheet Language Transformations* (расширяемый язык трансформации таблиц стилей). XSLT представляет собой язык XML, который описывает преобразование одного XML-текста в другой. Наиболее типичным примером такого преобразования служит трансформация XML-документа (который обычно описывает данные) в XHTML-документ (описывающий форматированный документ).

Рассмотрим следующий XML-файл:

```
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Показанный ниже XSLT-файл описывает такое преобразование:

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  version="1.0">
  <xsl:template match="/">
    <html>
      <p><xsl:value-of select="//firstname"/></p>
      <p><xsl:value-of select="//lastname"/></p>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Вывод выглядит так:

```
<html>
  <p>Jim</p>
  <p>Bo</p>
</html>
```

Класс `System.Xml.Xsl.XslCompiledTransform` эффективно выполняет XLST-преобразования. Он является заменой устаревшему классу `XmlTransform`. Класс `XmlTransform` работает очень просто:

```
XslCompiledTransform transform = new XslCompiledTransform();
transform.Load ("test.xslt");
transform.Transform ("input.xml", "output.xml");
```

Обычно удобнее пользоваться перегруженной версией метода `Transform`, которая вместо выходного файла принимает объект `XmlWriter`, что позволяет управлять форматированием.

# 12

# Освобождение и сборка мусора

Некоторые объекты требуют написания явного кода для освобождения таких ресурсов, как открытые файлы, блокировки, дескрипторы операционной системы и неуправляемые объекты. В терминологии .NET процедура называется *освобождением* и поддерживается через интерфейс `IDisposable`. Управляемая память, занятая неиспользуемыми объектами, также в какой-то момент должна быть возвращена; такая функция называется *сборкой мусора* и выполняется средой CLR.

Освобождение отличается от сборки мусора в том, что оно обычно инициируется явно; сборка мусора является полностью автоматической. Другими словами, программист заботится об освобождении файловых дескрипторов, блокировок и ресурсов операционной системы, а среда CLR занимается освобождением памяти.

В настоящей главе обсуждаются темы освобождения и сборки мусора, а также рассматриваются финализаторы C# и шаблон, согласно которому они могут предоставить страховку для освобождения. Наконец, здесь раскрываются тонкости сборщика мусора и другие варианты управления памятью.

## `IDisposable`, `Dispose` и `Close`

В .NET Framework определен специальный интерфейс для типов, требующих метода освобождения:

```
public interface IDisposable
{
  void Dispose();
}
```

Оператор `using` языка C# предлагает синтаксическое сокращение для вызова метода `Dispose` на объектах, которые реализуют интерфейс `IDisposable`, используя блок `try/finally`. Например:

```
using (FileStream fs = new FileStream ("myFile.txt", FileMode.Open))
{
  // ... Записать в файл ...
}
```

Компилятор преобразует такой код следующим образом:

```
FileStream fs = new FileStream ("myFile.txt", FileMode.Open);
try
{
  // ... Записать в файл ...
}
finally
{
  if (fs != null) ((IDisposable)fs).Dispose();
}
```

Блок `finally` гарантирует, что метод `Dispose` вызывается даже в случае генерации исключения[1] или принудительного раннего выхода из блока.

В простых сценариях создание собственного освобождаемого типа сводится просто к реализации интерфейса `IDisposable` и написанию метода `Dispose`:

```
sealed class Demo : IDisposable
{
  public void Dispose()
  {
    // Выполнить очистку/освобождение
    ...
  }
}
```

Такой шаблон хорошо работает в простых случаях и подходит для запечатанных классов. В разделе "Вызов метода `Dispose` из финализатора" далее в главе мы представим более продуманный шаблон, который может обеспечить страховку для потребителей, забывших вызвать `Dispose`. В случае незапечатанных типов есть веские основания следовать последнему шаблону с самого начала – иначе наступит крупная путаница, когда подтипы сами пожелают добавить функциональность подобного рода.

## Стандартная семантика освобождения

Логика освобождения в инфраструктуре .NET Framework подчиняется фактическому набору правил. Эти правила не являются жестко привязанными к .NET Framework или к языку C#; их назначение заключается в том, чтобы определить согласованный протокол для потребителей. Правила описаны далее.

1. После освобождения объект находится в "подвешенном" состоянии. Его нельзя реактивировать, а обращение к его методам или свойствам (отличным от `Dispose`) становится причиной генерации исключения `ObjectDisposedException`.
2. Многократный вызов метода `Dispose` объекта не приводит к ошибкам.
3. Если освобождаемый объект *x* "владеет" освобождаемым объектом *y*, то метод `Dispose` объекта *x* автоматически вызывает метод `Dispose` объекта *y* – при условии, что не указано иначе.

Перечисленные правила также полезны при написании собственных типов, хотя они не являются обязательными. Ничто не способно остановить вас от написания

[1] В разделе "`Interrupt` и `Abort`" главы 22 мы покажем, как прекращение работы потока может нарушать безопасность данного шаблона. На практике подобное редко является проблемой, потому что прерывать потоки настоятельно не рекомендуется именно по этой и другим причинам.

метода вроде "Undispose", разве только перспектива получить взбучку от коллег по разработке!

Согласно правилу 3 объект контейнера автоматически освобождает свои дочерние объекты. Хорошим примером может служить контейнерный элемент управления Windows, такой как `Form` или `Panel`. Контейнер может размещать множество дочерних элементов управления, однако вы не должны освобождать каждый из них явно: обо всех них позаботится закрываемый или освобождаемый родительский элемент управления либо форма. Еще один пример – помещение типа `FileStream` в оболочку `DeflateStream`. Освобождение `DeflateStream` также приводит к освобождению `FileStream` – если только в конструкторе не указано иное.

### `Close` и `Stop`

Некоторые типы в дополнение к `Dispose` определяют метод по имени `Close`. Платформа .NET Framework не полностью согласована относительно семантики метода `Close`, хотя почти во всех случаях он обладает одной из следующих характеристик:

- является функционально идентичным методу `Dispose`;
- реализует *подмножество* функциональности метода `Dispose`.

Второй характеристикой обладает, например, интерфейс `IDbConnection`: подключение с состоянием `Closed` (закрыто) можно повторно открыть посредством метода `Open`, но сделать это для освобожденного (вызовом `Dispose`) подключения нельзя. Другим примером может быть Windows-элемент `Form`, активизированный с помощью метода `ShowDialog`: вызов `Close` скрывает его, а вызов `Dispose` освобождает его ресурсы.

В некоторых классах определен метод `Stop` (скажем, в `Timer` и `HttpListener`). Метод `Stop` может освобождать неуправляемые ресурсы подобно `Dispose`, но в отличие от `Dispose` он разрешает последующий перезапуск (посредством `Start`).

В WinRT метод `Close` рассматривается как идентичный `Dispose` – на самом деле исполняющая среда *проецирует* методы, называемые `Close`, на методы, называемые `Dispose`, чтобы сделать их типы дружественными к операторам `using`.

## Когда выполнять освобождение

Безопасное правило, которому нужно следовать (почти во всех случаях), формулируется так: если есть сомнения, тогда необходимо освобождать. Освобождаемый объект (если бы он мог разговаривать) сказал бы следующее.

> *Когда вы завершите работать со мной, уведомьте меня. Если просто так меня оставить, то могут возникнуть проблемы с экземплярами других объектов, доменом приложения, компьютером, сетью или базой данных!*

Объекты, содержащие неуправляемый дескриптор ресурса, почти всегда требуют освобождения, чтобы освободить такой дескриптор. Примеры включают элементы управления Windows Forms, файловые или сетевые потоки, сетевые сокеты, перья, кисти и растровые изображения GDI+. И наоборот, если тип является освобождаемым, тогда он часто (но не всегда) ссылается на неуправляемый дескриптор, прямо или косвенно. Причина в том, что неуправляемые дескрипторы предоставляют шлюз во "внешний мир" ресурсам операционной системы, сетевым подключениям, блокировкам базы данных – основным средствам, из-за некорректного отбрасывания которых объекты могут создавать проблемы за своими пределами.

Тем не менее, существуют три сценария, когда освобождение *не* нужно:

- когда вы не "владеете" объектом, например, при получении *разделяемого* объекта через статическое поле или свойство;
- когда метод `Dispose` объекта выполняет какое-то нежелательное действие;
- когда метод `Dispose` объекта является лишним *согласно проекту*, и освобождение такого объекта добавляет сложности программе.

Первый сценарий встречается редко. Основные случаи отражены в пространстве имен `System.Drawing`: объекты GDI+, получаемые через *статические поля или свойства* (такие как `Brushes.Blue`), никогда не должны освобождаться, поскольку один и тот же экземпляр задействован на протяжении всего времени жизни приложения. Однако экземпляры, которые получаются с помощью конструкторов (скажем, через `new SolidBrush`), *должны* быть освобождены, как и должны освобождаться экземпляры, полученные посредством статических *методов* (вроде `Font.FromHdc`).

Второй сценарий более распространен. В пространствах имен `System.IO` и `System.Data` можно найти ряд удачных примеров.

| Тип | Что делает функция освобождения | Когда освобождение выполнять не нужно |
|---|---|---|
| `MemoryStream` | Предотвращает дальнейший ввод-вывод | Когда позже необходимо читать/записывать в поток |
| `StreamReader`, `StreamWriter` | Сбрасывает средство чтения/записи и закрывает лежащий в основе поток | Когда лежащий в основе поток должен быть сохранен открытым (взамен по окончании потребуется вызвать метод `Flush` на объекте `StreamWriter`) |
| `IDbConnection` | Освобождает подключение к базе данных и очищает строку подключения | Если необходимо повторно открыть его с помощью `Open`, должен быть вызван метод `Close`, а не `Dispose` |
| `DataContext` (LINQ to SQL) | Предотвращает дальнейшее использование | Когда могут существовать лениво оцениваемые запросы, подключенные к данному контексту |

Метод `Dispose` класса `MemoryStream` делает недоступным только сам объект; он не выполняет никакой критически важной очистки, потому что `MemoryStream` не удерживает неуправляемые дескрипторы или другие ресурсы подобного рода.

Третий сценарий охватывает следующие классы: `WebClient`, `StringReader`, `StringWriter` и `BackgroundWorker` (из пространства имен `System.ComponentModel`). Упомянутые типы являются освобождаемыми по принуждению их базового класса, а не по причине реальной потребности в выполнении необходимой очистки. Если приходится создавать и работать с таким объектом полностью внутри одного метода, тогда помещение его в блок `using` привносит лишь небольшое неудобство. Но если объект является более долговечным, то процедура выяснения, когда он больше не применяется и может быть освобожден, добавляет излишнюю сложность. В таких случаях можно просто проигнорировать освобождение объекта.

Игнорирование освобождения может иногда повлечь за собой снижение производительности (см. раздел "Вызов метода Dispose из финализатора" далее в главе).

## Подключаемое освобождение

Поскольку интерфейс IDisposable позволяет типу обрабатываться конструкцией using в C#, возникает соблазн расширить охват IDisposable на несущественные действия. Например:

```
public sealed class HouseManager : IDisposable
{
  public void Dispose()
  {
    CheckTheMail();
  }
  ...
}
```

Идея в том, что потребитель данного класса может решить обойти несущественную очистку, просто не вызывая Dispose. Однако это опирается на знание потребителем о том, что именно находится внутри метода Dispose класса HouseManager. Кроме того, работа будет нарушена, если позже добавится действие *существенной* очистки:

```
public void Dispose()
{
  CheckTheMail();  // Несущественная очистка
  LockTheHouse();  // Существенная очистка
}
```

Решение такой проблемы предлагает шаблон подключаемого освобождения:

```
public sealed class HouseManager : IDisposable
{
  public readonly bool CheckMailOnDispose;

  public HouseManager (bool checkMailOnDispose)
  {
    CheckMailOnDispose = checkMailOnDispose;
  }

  public void Dispose()
  {
    if (CheckMailOnDispose) CheckTheMail();
    LockTheHouse();
  }
  ...
}
```

Потребитель всегда может впоследствии вызвать метод Dispose, обеспечивая простоту и устраняя необходимость в написании специальной документации или в проведении рефлексии. Примером реализации шаблона является класс DeflateStream из пространства имен System.IO.Compression. Вот его конструктор:

```
public DeflateStream (Stream stream, CompressionMode mode, bool leaveOpen)
```

Несущественное действие связано с закрытием внутреннего потока (параметр stream) при освобождении. Временами внутренний поток нужно оставлять откры-

тым и по-прежнему освобождать объект DeflateStream, чтобы выполнялось его *существенное* действие освобождения (сбрасывание буферизированных данных).

Шаблон может выглядеть простым, но до выхода .NET Framework 4.5 он отсутствовал в классах StreamReader и StreamWriter (из пространства имен System.IO). Результатом стало неаккуратное решение: класс StreamWriter вынужден был открывать доступ к еще одному методу (Flush) для выполнения существенной очистки потребителями, не вызывающими Dispose. (В .NET Framework 4.5 для указанных классов теперь доступен конструктор, который позволяет удерживать поток открытым.) Класс CryptoStream в пространстве имен System.Security.Cryptography страдает от похожей проблемы и требует вызова метода FlushFinalBlock для своего освобождения с сохранением внутреннего потока открытым.

Это можно было бы описать как проблему *владения*. Освобождаемый объект должен ответить на вопрос: действительно ли он владеет внутренним ресурсом, который в нем задействован? Или же ресурс просто арендуется у кого-то еще, кто управляет как временем жизни внутреннего ресурса, так и (посредством недокументированного контракта) временем жизни освобождаемого объекта?

Следование шаблону подключаемого освобождения позволяет избежать данной проблемы, делая контракт владения документированным и явным.

## Очистка полей при освобождении

В общем случае очищать поля объекта в его методе Dispose вовсе не обязательно. Тем не менее, рекомендуемой практикой является отмена подписки на события, на которые объект был подписан внутренне во время своего существования (пример приведен в разделе "Утечки управляемой памяти" далее в главе). Отмена подписки на события подобного рода позволяет избежать получения нежелательных уведомлений о событиях, а также непреднамеренного сохранения объекта в активном состоянии с точки зрения сборщика мусора (garbage collector – GC).

Сам по себе метод Dispose не вызывает освобождения (управляемой) памяти – это может произойти только при сборке мусора.

Стоит также установить какое-то поле, указывающее на факт освобождения объекта, чтобы можно было сгенерировать исключение ObjectDisposedException, если потребитель позже попытается обратиться к членам освобожденного объекта. Хороший шаблон предусматривает использование свойства, открытого для чтения:

```
public bool IsDisposed { get; private set; }
```

Хотя формально и необязательно, но неплохо также очистить собственные обработчики событий объекта (устанавливая их в null) в методе Dispose. Тем самым исключается возможность возникновения таких событий во время или после освобождения.

Иногда объект хранит ценную секретную информацию, такую как ключи шифрования. В подобных случаях имеет смысл во время освобождения очистить эти данные в полях (во избежание их обнаружения менее привилегированными сборками или вредоносным программным обеспечением). Именно так поступает класс SymmetricAlgorithm в пространстве имен System.Security.Cryptography, вызывая метод Array.Clear на байтовом массиве, который хранит ключ шифрования.

# Автоматическая сборка мусора

Независимо от того, требует ли объект метода `Dispose` для специальной логики освобождения, в какой-то момент память, занимаемая им в куче, должна быть освобождена. Среда CLR обрабатывает данный аспект полностью автоматически посредством автоматического сборщика мусора. Вы никогда не освобождаете управляемую память самостоятельно. Например, взгляните на следующий метод:

```
public void Test()
{
  byte[] myArray = new byte[1000];
  ...
}
```

Когда метод `Test` выполняется, в куче выделяется память под массив для удержания 1000 байтов. Ссылка на массив осуществляется через локальную переменную `myArray`, хранящуюся в стеке. Когда метод завершается, локальная переменная `myArray` покидает область видимости, а это значит, что ничего не остается для ссылки на массив в куче. Висячий массив затем может быть утилизирован при сборке мусора.

В режиме отладки с отключенной оптимизацией время жизни объекта, на который производится ссылка с помощью локальной переменной, расширяется до конца блока кода, чтобы упростить процесс отладки. В противном случае объект становится пригодным для сборки мусора в самой ранней точке, после которой им перестали пользоваться.

Сборка мусора не происходит немедленно после того, как объект становится висячим. Почти как уборка мусора на улицах, она выполняется периодически, хотя (в отличие от уборки улиц) и не по фиксированному графику. Среда CLR основывает свое решение о том, когда инициировать сборку мусора, на ряде факторов, таких как доступная память, объем выделенной памяти и время, прошедшее с последней сборки. Это значит, что между моментом, когда объект становится висячим, и моментом, когда занятая им память будет освобождена, имеется неопределенная задержка. Такая задержка может варьироваться в пределах от наносекунд до дней.

Сборщик мусора не собирает весь мусор при каждой сборке. Взамен диспетчер памяти разделяет объекты на *поколения*, и сборщик мусора выполняет сборку новых поколений (недавно распределенных объектов) чаще, чем старых поколений (объектов, существующих на протяжении длительного времени). Мы обсудим принцип более подробно в разделе "Как работает сборщик мусора?" далее в главе.

## Сборка мусора и потребление памяти

Сборщик мусора старается соблюдать баланс между временем, затрачиваемым на сборку мусора, и потреблением памяти со стороны приложения (рабочим набором). Следовательно, приложения могут расходовать больше памяти, чем им необходимо, особенно если конструируются крупные временные массивы.

Отслеживать потребление памяти процессом можно с помощью диспетчера задач Windows или монитора ресурсов – либо программно запрашивая счетчик производительности:

```
// Эти типы находятся в пространстве имен System.Diagnostics:
string procName = Process.GetCurrentProcess().ProcessName;
using (PerformanceCounter pc = new PerformanceCounter
     ("Process", "Private Bytes", procName))
  Console.WriteLine (pc.NextValue());
```

В данном коде запрашивается *закрытый рабочий набор*, который дает наилучшую общую картину потребления памяти программой. В частности, он исключает память, которую среда CLR освободила внутренне и готова возвратить операционной системе, если в данной памяти нуждается другой процесс.

## Корневые объекты

Корневой объект – это то, что сохраняет определенный объект в активном состоянии. Если на какой-то объект нет прямой или косвенной ссылки со стороны корневого объекта, то он будет пригоден для сборки мусора.

Корневым объектом может выступать одна из следующих сущностей:

- локальная переменная или параметр в выполняющемся методе (или в любом методе внутри его стека вызовов);
- статическая переменная;
- объект в очереди, которая хранит объекты, готовые к финализации (см. следующий раздел).

Код в удаленном объекте не может выполняться, а потому если есть хоть какая-то вероятность выполнения некоторого метода (экземпляра), то на его объект должна существовать ссылка одним из указанных выше способов.

Обратите внимание, что группа объектов, которые циклически ссылаются друг на друга, считается висячей, если отсутствует ссылка из корневого объекта (рис. 12.1). Выражаясь по-другому, объекты, которые не могут быть доступны за счет следования по стрелкам (ссылкам) из корневого объекта, являются *недостижимыми* и таким образом подпадают под сборку мусора.

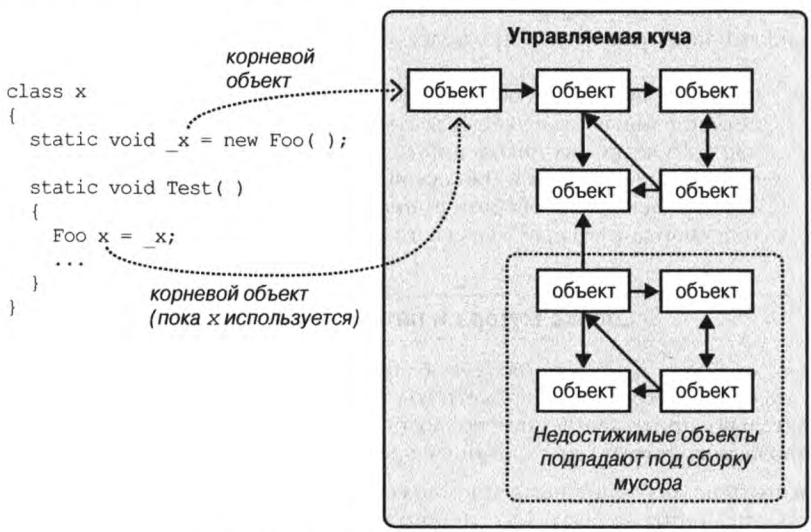



***Рис. 12.1.** Пример корневых объектов*

## Сборка мусора и WinRT

При освобождении памяти WinRT полагается на механизм подсчета ссылок COM, а не на автоматический сборщик мусора. Несмотря на это, объекты WinRT, которые создаются в коде C#, имеют время жизни, управляемое сборщиком мусора CLR. Причина в том, что для доступа к COM-объекту среда CLR использует посредник – объект, который создается "за кулисами" и называется *вызываемой оболочкой времени выполнения* (runtime callable wrapper – RCW), как будет показано в главе 24.

# Финализаторы

Перед освобождением объекта из памяти запускается его *финализатор*, если он предусмотрен. Финализатор объявляется подобно конструктору, но его имя снабжается префиксом ~:

```
class Test
{
  ~Test()
  {
    // Логика финализатора...
  }
}
```

(Несмотря на сходство в объявлении с конструктором, финализаторы не могут быть объявлены как `public` или `static`, не могут принимать параметры и не могут обращаться к базовому классу.)

Финализаторы возможны потому, что работа сборки мусора организована в виде отличающихся фаз. Первым делом сборщик мусора идентифицирует неиспользуемые объекты, готовые к удалению. Те из них, которые не имеют финализаторов, удаляются сразу. Те из них, которые располагают отложенными (незапущенными) финализаторами, сохраняются в активном состоянии (на текущий момент) и помещаются в специальную очередь.

В данной точке сборка мусора завершена, и программа продолжает выполнение. Затем параллельно программе начинает выполняться *поток финализаторов*, который выбирает объекты из этой специальной очереди и запускает их методы финализации. Перед запуском финализатора каждый объект по-прежнему активен – специальная очередь действует в качестве корневого объекта. После того, как объект извлечен из очереди, а его финализатор выполнен, объект становится висячим и будет удален при следующей сборке мусора (для данного поколения объекта).

Финализаторы могут быть удобными, но с рядом оговорок.

- Финализаторы замедляют выделение и утилизацию памяти (сборщик мусора должен отслеживать, какие финализаторы были запущены).
- Финализаторы продлевают время жизни объекта и любых объектов, которые на него *ссылаются* (они вынуждены ожидать действительного удаления при очередной сборке мусора).
- Порядок вызова финализаторов для набора объектов предсказать невозможно.
- Имеется только ограниченный контроль над тем, когда будет вызван финализатор того или иного объекта.
- Если код в финализаторе приводит к блокировке, то другие объекты не смогут выполнить финализацию.
- Финализаторы могут вообще не запуститься, если приложение не смогло выгрузиться чисто.

В целом финализаторы кое в чем похожи на юристов – хотя и бывают ситуации, когда они действительно нужны, обычно прибегать к их услугам желания нет, разве что в случае крайней необходимости. К тому же, если вы решили воспользоваться услугами юристов, то должны быть абсолютно уверены в понимании того, что они делают для вас.

Ниже приведены некоторые руководящие принципы, применяемые при реализации финализаторов.

- Удостоверьтесь, что финализатор выполняется быстро.
- Никогда не блокируйте финализатор (глава 14).
- Не ссылайтесь на другие финализируемые объекты.
- Не генерируйте исключения.

Финализатор объекта может быть вызван, даже если во время конструирования сгенерировано исключение. По этой причине при написании финализатора лучше не предполагать, что все поля были корректно инициализированы.

## Вызов метода `Dispose` из финализатора

Популярный шаблон предусматривает вызов в финализаторе метода `Dispose`. Подобное действие имеет смысл, когда очистка не является срочной, и ее ускорение вызовом метода `Dispose` является больше оптимизацией, нежели необходимостью.

Имейте в виду, что в данном шаблоне вы объединяете вместе освобождение памяти и освобождение ресурсов – две вещи с потенциально несовпадающими интересами (если только сам ресурс не является памятью). Вы также увеличиваете нагрузку на поток финализации.

Такой шаблон также может использоваться в качестве страховки для случаев, когда потребитель попросту забывает вызвать метод `Dispose`. Однако затем подобную небрежность неплохо бы зарегистрировать в журнале, чтобы впоследствии исправить ошибку.

Ниже показан стандартный шаблон реализации:

```
class Test : IDisposable
{
  public void Dispose()            // НЕ virtual
  {
    Dispose (true);
    GC.SuppressFinalize (this);    // Препятствует запуску финализатора
  }

  protected virtual void Dispose (bool disposing)
  {
    if (disposing)
    {
      // Вызвать метод Dispose на других объектах,
      // которыми владеет данный экземпляр.
      // Здесь можно ссылаться на другие финализируемые объекты.
      // ...
    }
```

```
    // Освободить неуправляемые ресурсы, которыми владеет (только) этот объект.
    // ...
  }
  ~Test()
  {
    Dispose (false);
  }
}
```

Метод `Dispose` перегружен для приема флага `disposing` типа `bool`. Версия без параметров *не* объявлена как `virtual` и просто вызывает расширенную версию `Dispose` с передачей ей значения `true`.

Расширенная версия содержит действительную логику освобождения и помечена как `protected` и `virtual`, предоставляя подклассам безопасную точку для добавления собственной логики освобождения. Флаг `disposing` означает, что он вызывается "подходящим образом" из метода `Dispose`, а не в "режиме крайнего средства" из финализатора. Идея в том, что при вызове с флагом `disposing`, установленным в `false`, этот метод в общем случае не должен ссылаться на другие объекты с финализаторами (поскольку такие объекты могут сами быть финализированными и, таким образом, находиться в непредсказуемом состоянии). Как видите, правила исключают довольно многое! Существует пара задач, которые по-прежнему могут выполняться в режиме крайнего средства, когда `disposing` равно `false`:

- освобождение любых *прямых ссылок* на ресурсы операционной системы (полученных возможно через обращение P/Invoke к Win32 API);
- удаление временного файла, созданного при конструировании.

Чтобы сделать описанный подход надежным, любой код, способный сгенерировать исключение, должен быть помещен в блок `try/catch`, а исключение в идеальном случае необходимо регистрировать в журнале. Любая регистрация в журнале должна быть насколько возможно простой и надежной.

Обратите внимание, что внутри метода `Dispose` без параметров мы вызываем метод `GC.SuppressFinalize` — это предотвращает запуск финализатора, когда сборщик мусора позже доберется до него. Формально подобное необязательно, т.к. методы `Dispose` должны допускать многократные вызовы. Тем не менее, такой прием повышает производительность, потому что позволяет подвергнуть данный объект (и объекты, которые на него ссылаются) процедуре сборки мусора в единственном цикле.

## Восстановление

Предположим, что финализатор модифицирует активный объект так, что он снова ссылается на неактивный объект. Во время очередной сборки мусора (для поколения объекта) среда CLR выяснит, что ранее неактивный объект больше не является висячим, поэтому он должен избежать сборки мусора. Такой расширенный сценарий называется *восстановлением*.

В целях иллюстрации предположим, что нужно написать класс, который управляет временным файлом. Во время сборки мусора для экземпляра данного класса финализатор класса должен удалить временный файл. Решение задачи кажется простым:

```
public class TempFileRef
{
  public readonly string FilePath;
  public TempFileRef (string filePath) { FilePath = filePath; }
  ~TempFileRef() { File.Delete (FilePath); }
}
```

К сожалению, здесь присутствует ошибка: вызов метода `File.Delete` может сгенерировать исключение (возможно, из-за нехватки разрешений, по причине того, что файл в текущий момент используется, или файл уже был удален). Такое исключение привело бы к нарушению работы всего приложения (и воспрепятствовало бы запуску других финализаторов). Мы могли бы просто "поглотить" исключение с помощью пустого блока перехвата, но тогда не было бы известно, что именно пошло не так. Обращение к некоторому хорошо продуманному API-интерфейсу сообщения об ошибках также нежелательно, поскольку это привнесет накладные расходы в поток финализаторов, затрудняя проведение сборки мусора для других объектов. Мы хотим, чтобы действия финализации были простыми, надежными и быстрыми.

Более удачное решение предполагает запись информации об отказе в статическую коллекцию:

```
public class TempFileRef
{
  static ConcurrentQueue<TempFileRef> _failedDeletions
    = new ConcurrentQueue<TempFileRef>();

  public readonly string FilePath;
  public Exception DeletionError { get; private set; }

  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef()
  {
    try { File.Delete (FilePath); }
    catch (Exception ex)
    {
      DeletionError = ex;
      _failedDeletions.Enqueue (this);  // Восстановление
    }
  }
}
```

Занесение объекта в статическую коллекцию `_failedDeletions` предоставляет ему еще одну ссылку, гарантируя тем самым, что он останется активным до тех пор, пока со временем не будет изъят из `_failedDeletions`.

Класс `ConcurrentQueue<T>` является безопасной к потокам версией класса `Queue<T>` и определен в пространстве имен `System.Collections.Concurrent` (глава 23). Коллекция, безопасная к потокам, применяется по двум причинам. Во-первых, среда CLR резервирует право на выполнение финализаторов более чем одному потоку параллельно. Это значит, что при доступе к разделяемому состоянию, такому как статическая коллекция, мы должны учитывать возможность одновременной финализации двух объектов. Во-вторых, в определенный момент понадобится изъять элементы из `_failedDeletions`, чтобы предпринять в отношении них какие-то действия. Действия должны выполняться также в безопасной к потокам манере, потому что они могут происходить одновременно с занесением в коллекцию другого объекта финализатором.

### GC.ReRegisterForFinalize

Финализатор восстановленного объекта не запустится во второй раз, если только не вызвать метод `GC.ReRegisterForFinalize`.

В следующем примере мы пытаемся удалить временный файл внутри финализатора (как в последнем примере). Но если удаление терпит неудачу, то мы повторно регистрируем объект, чтобы предпринять новую попытку при следующей сборке мусора:

```
public class TempFileRef
{
  public readonly string FilePath;
  int _deleteAttempt;

  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef()
  {
    try { File.Delete (FilePath); }
    catch
    {
      if (_deleteAttempt++ < 3) GC.ReRegisterForFinalize (this);
    }
  }
}
```

После третьей неудавшейся попытки финализатор молча отказывается от удаления файла. Мы могли бы расширить такое поведение, скомбинировав его с предыдущим примером — другими словами, после третьего отказа добавить объект в очередь `_failedDeletions`.

Позаботьтесь о том, чтобы метод `ReRegisterForFinalize` вызывался в финализаторе только один раз. Двукратный вызов приведет к тому, что объект будет перерегистрирован дважды и ему придется пройти две дополнительных финализации!

# Как работает сборщик мусора?

Стандартная среда CLR использует сборщик мусора с поддержкой поколений, пометки и сжатия, который осуществляет автоматическое управление памятью для объектов, хранящихся в управляемой куче. Сборщик мусора считается *отслеживающим* в том, что он не вмешивается в каждый доступ к объекту, а взамен активизируется периодически и отслеживает граф объектов, хранящихся в управляемой куче, с целью определения объектов, которые могут расцениваться как мусор и потому подвергаться сборке.

Сборщик мусора инициирует процесс сборки при выделении памяти (через ключевое слово `new`) либо после того, как выделенный объем памяти превысил определенный порог, либо в другие моменты, чтобы уменьшить объем памяти, занимаемой приложением. Процесс сборки можно также активизировать вручную, вызвав метод `System.GC.Collect`. Во время сборки мусора все потоки могут быть заморожены (более подробно об этом рассказывается в следующем разделе).

Сборщик мусора начинает со ссылок на корневые объекты и проходит по графу объектов, помечая все затрагиваемые им объекты как достижимые. После завершения процесса все объекты, которые не были помечены, считаются неиспользуемыми и пригодными к сборке мусора.

Неиспользуемые объекты без финализаторов отбрасываются немедленно, а неиспользуемые объекты с финализаторами помещаются в очередь для обработки потоком финализаторов после завершения сборщика мусора. Эти объекты затем становятся пригодными к сборке при следующем запуске сборщика мусора для данного поколения объектов (если только они не будут восстановлены).

Оставшиеся активные объекты затем смещаются в начало кучи (сжимаются), освобождая пространство под дополнительные объекты. Сжатие служит двум целям: оно устраняет фрагментацию памяти и позволяет сборщику мусора при выделении памяти под новые объекты применять очень простую стратегию – всегда выделять память в конце кучи. Подобный подход позволяет избежать выполнения потенциально длительной задачи по ведению списка сегментов свободной памяти.

Если оказывается, что после сборки мусора памяти для размещения нового объекта недостаточно, и операционная система не может выделить дополнительную память, тогда генерируется исключение `OutOfMemoryException`.

## Приемы оптимизации

Для сокращения времени сборки мусора в сборщике предпринимаются разнообразные приемы оптимизации.

### Сборка с учетом поколений

Самая важная оптимизация связана с тем, что сборщик мусора поддерживает поколения. Концепция поколений извлекает преимущества из того факта, что хотя многие объекты распределяются и быстро отбрасываются, некоторые объекты существуют длительное время, поэтому не должны отслеживаться в течение каждой сборки.

По существу сборщик мусора разделяет управляемую кучу на три поколения. Объекты, которые были только что распределены, относятся к поколению *Gen0*, объекты, выдержавшие один цикл сборки – к поколению *Gen1*, а все остальные объекты – к поколению *Gen2*. Поколения Gen0 и Gen1 называются *недолговечными*.

Среда CLR сохраняет раздел Gen0 относительно небольшим (максимум 256 Мбайт в 64-разрядной версии CLR для рабочей станции; с типичным размером от сотен килобайтов до нескольких мегабайтов). Когда раздел Gen0 заполняется, сборщик мусора GC вызывает сборку Gen0 – что происходит относительно часто. Сборщик мусора применяет похожий порог памяти к разделу Gen1 (который действует в качестве буфера для Gen2), поэтому сборки Gen1 тоже являются относительно быстрыми и частыми. Однако полные сборки мусора, включающие Gen2, занимают намного больше времени и в итоге происходят нечасто. Результат полной сборки мусора показан на рис. 12.2.

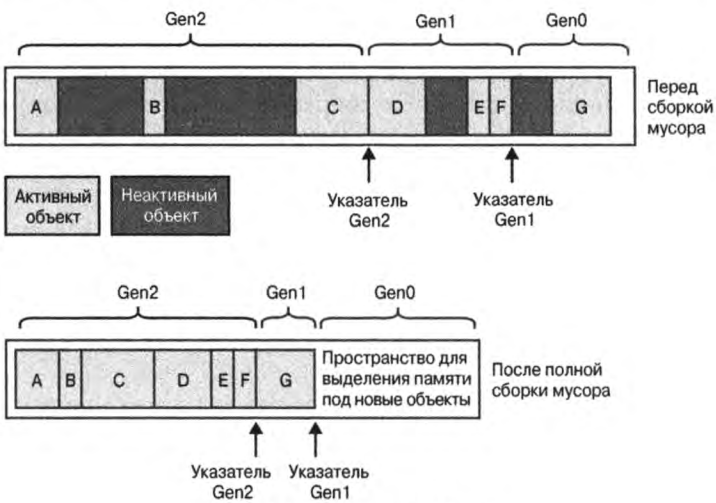



***Рис. 12.2.** Поколения кучи.*

Вот очень приблизительные цифры: сборка Gen0 может занимать менее 1 миллисекунды, так что заметить ее в типовом приложении нереально. Но полная сборка мусора в программе с крупными графами объектов может длиться примерно 100 миллисекунд. Цифры зависят от множества факторов и могут значительно варьироваться – особенно в случае раздела Gen2, размер которого *не ограничен* (в отличие от разделов Gen0 и Gen1).

В результате кратко живущие объекты очень эффективны в своем использовании сборщика мусора. Экземпляры `StringBuilder`, создаваемые в следующем методе, почти наверняка будут собраны при быстрой сборке Gen0:

```
string Foo()
{
  var sb1 = new StringBuilder ("test");
  sb1.Append ("...");
  var sb2 = new StringBuilder ("test");
  sb2.Append (sb1.ToString());
  return sb2.ToString();
}
```

### Куча для массивных объектов

Для объектов, размеры которых превышают определенный порог (в настоящее время составляющий 85 000 байтов), сборщик мусора применяет отдельную область, которая называется *кучей для массивных объектов* (Large Object Heap – LOH). Это позволяет избежать избыточных сборок Gen0 – в отсутствие LOH распределение последовательности объектов с размерами в 16 Мбайт могло бы приводить к запуску сборки Gen0 после каждого распределения.

По умолчанию область LOH не сжимается, поскольку перемещение крупных блоков памяти во время сборки мусора будет чрезмерно затратным. Отсюда два последствия.

- Выделение памяти может стать медленнее, т.к. сборщик мусора не всегда способен просто распределять объекты в конце кучи – он должен также искать промежутки в середине, что требует поддержки связного списка свободных блоков памяти[2].
- Область LOH подвержена *фрагментации*. Это значит, что освобождение объекта может привести к возникновению дыры в LOH, которую впоследствии трудно заполнить. Например, дыра, оставленная 86000-байтовым объектом, может быть заполнена только объектом с размером между 85 000 и 86 000 байтов (если только рядом не примыкает еще одна дыра).

В случаях, когда описанные последствия могут приводить к проблемам, сборщик мусора можно проинструктировать о необходимости сжатия области LOH при следующей сборке:

```
GCSettings.LargeObjectHeapCompactionMode =
  GCLargeObjectHeapCompactionMode.CompactOnce;
```

Куча для массивных объектов не поддерживает концепцию поколений: все объекты трактуются как относящиеся к поколению Gen2.

[2] То же самое может иногда возникать в куче с поколениями из-за закрепления (см. раздел "Оператор `fixed`" в главе 4).

### Параллельная и фоновая сборка мусора

Сборщик мусора должен заморозить (заблокировать) потоки выполнения на периоды проведения сборки мусора. Сюда входит весь период, в течение которого производится сборка Gen0 или Gen1.

Тем не менее, сборщик мусора прикладывает специальные усилия, чтобы предоставить потокам возможность выполняться во время сборки Gen2, т.к. замораживание приложения на потенциально длительный период нежелательно. Оптимизация подобного рода применяется только к версии CLR для рабочей станции, которая используется в настольных компьютерах, функционирующих под управлением Windows (и для всех версий Windows с автономными приложениями). Логическое обоснование заключается в том, что задержка от блокирующей сборки мусора менее вероятно окажется проблемой в серверных приложениях, не имеющих пользовательского интерфейса.

Смягчающим фактором является то, что серверная версия CLR при сборке мусора задействует все свободные процессорные ядра, так что сервер с восьмиядерным процессором будет выполнять полную сборку мусора во много раз быстрее. На самом деле серверный сборщик мусора настроен на максимизацию полосы пропускания, а не на минимизацию задержки.

Оптимизация на стороне рабочей станции ранее называлась *параллельной сборкой мусора*. В версии CLR 4.0 она была модернизирована и переименована в *фоновую сборку мусора*. Фоновая сборка мусора устраняет ограничение, заключающееся в том, что параллельная сборка мусора перестает быть параллельной, если раздел Gen0 заполнился, пока выполнялась сборка Gen2. Таким образом, начиная с версии CLR 4.0, приложения, которые постоянно выделяют память, будут более отзывчивыми.

### Уведомления о сборке мусора (серверная версия CLR)

Серверная версия CLR может отправлять уведомления непосредственно перед началом полной сборки мусора. Это предназначено для конфигураций ферм серверов: идея состоит в переадресации запросов другим серверам прямо перед началом сборки мусора. Затем немедленно инициируется сборка мусора и производится ожидание ее завершения перед переадресацией запросов обратно данному серверу.

Чтобы начать выдачу уведомлений, необходимо вызвать метод `GC.RegisterForFullGCNotification`. Далее потребуется запустить другой поток (глава 14), в котором сначала вызывается метод `GC.WaitForFullGCApproach`. Когда этот метод возвратит значение перечисления `GCNotificationStatus`, указывающее на то, что сборка мусора уже близко, можно переадресовать запросы другим серверам и принудительно запустить сборку мусора вручную (см. следующий раздел). Затем следует вызвать метод `GC.WaitForFullGCComplete`: по возвращению управления из данного метода сборка мусора завершена, и можно снова принимать запросы. После этого весь цикл повторяется.

## Принудительный запуск сборки мусора

Принудительно запустить сборку мусора можно в любой момент, вызвав метод `GC.Collect`. Вызов `GC.Collect` без аргумента инициирует полную сборку мусора. Если передать целочисленное значение, тогда сборка выполнится для поколений, начиная с Gen0 и заканчивая поколением, номер которого соответствует указанному значению; таким образом, `GC.Collect(0)` выполняет только быструю сборку Gen0.

Обычно наилучшие показатели производительности можно получить, позволив сборщику мусора самостоятельно решать, что именно собирать: принудительная сборка мусора может нанести ущерб производительности за счет излишнего перевода объектов поколения Gen0 в поколение Gen1 (и объектов поколения Gen1 в поколение Gen2). Принудительная сборка также нарушит возможность *самонастройки* сборщика мусора, посредством которой сборщик динамически регулирует пороги для каждого поколения, чтобы добиться максимальной производительности во время работы приложения.

Тем не менее, существуют два исключения. Наиболее распространенным сценарием для вмешательства является ситуация, когда приложение собирается перейти в режим сна на некоторое время: хорошим примером может быть Windows-служба, которая выполняет ежесуточное действие (скажем, проверку обновлений). Такое приложение может использовать объект `System.Timers.Timer` для запуска действия каждые 24 часа. После завершения действия никакой другой код в течение 24 часов выполняться не будет, т.е. в данный период выделение памяти не делается и сборщик мусора не имеет шансов быть активизированным. Сколько бы памяти служба не потребила во время выполнения своего действия, память продолжит быть занятой в течение следующих 24 часов даже при пустом графе объектов! Решение предусматривает вызов метода `GC.Collect` сразу после завершения ежесуточного действия.

Чтобы обеспечить сборку мусора в отношении объектов, для которых она отложена финализаторами, можно предпринять дополнительный шаг – вызвать метод `WaitForPendingFinalizers` и запустить сборщик мусора еще раз:

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

Часто это делается в цикле: действие по выполнению финализаторов может освободить больше объектов, а не только те, которые имеют финализаторы.

Еще один сценарий для вызова метода `GC.Collect` связан с тестированием класса, располагающего финализатором.

## Настройка сборки мусора

Статическое свойство `GCSettings.LatencyMode` определяет способ, которым сборщик мусора балансирует задержку и общую эффективность. Изменение значения данного свойства со стандартного `Interactive` на `LowLatency` указывает среде CLR на необходимость применения более быстрых (но более частых) процедур сборки мусора. Это полезно, если приложение нуждается в очень быстром реагировании на события, возникающие в реальном времени.

Начиная с .NET Framework 4.6, сборщику мусора можно также сообщать о том, что процесс сборки должен быть временно приостановлен, вызывая метод `GC.TryStartNoGCRegion`, и затем возобновлять его с помощью метода `GC.EndNoGCRegion`.

## Нагрузка на память

Исполняющая среда решает, когда инициировать сборку мусора, на основе нескольких факторов, в числе которых общая загрузка памяти на машине. Если программа выделяет неуправляемую память (глава 25), то исполняющая среда получит нереалистично оптимистическое представление об использовании памяти программой, потому что среде CLR известно только об управляемой памяти. Чтобы ослабить

такое влияние, можно сообщить среде CLR о необходимости *учесть* выделение указанного объема неуправляемой памяти, вызвав метод `GC.AddMemoryPressure`. Чтобы отменить это (когда неуправляемая память освобождена), потребуется вызвать метод `GC.RemoveMemoryPressure`.

# Утечки управляемой памяти

В неуправляемых языках вроде C++ вы должны не забывать об освобождении памяти вручную, когда объект больше не требуется; в противном случае возникнет *утечка памяти*. В мире управляемых языков такая ошибка невозможна, поскольку в среде CLR существует система автоматической сборки мусора.

Несмотря на это, крупные и сложные приложения .NET могут демонстрировать аналогичный синдром в легкой форме с тем же самым конечным результатом: с течением времени жизни приложение потребляет все больше и больше памяти до тех пор, пока его не придется перезапустить. Хорошая новость в том, что утечки управляемой памяти обычно легче диагностировать и предотвращать.

Утечки управляемой памяти вызваны неиспользуемыми объектами, которые остаются активными по причине существования неиспользуемых или забытых ссылок на них. Распространенным кандидатом являются обработчики событий – они удерживают ссылку на целевой объект (если только он не является статическим методом). Например, взгляните на следующие классы:

```
class Host
{
  public event EventHandler Click;
}

class Client
{
  Host _host;
  public Client (Host host)
  {
    _host = host;
    _host.Click += HostClicked;
  }

  void HostClicked (object sender, EventArgs e) { ... }
}
```

Приведенный ниже тестовый класс содержит метод, который создает 1000 экземпляров класса `Client`:

```
class Test
{
  static Host _host = new Host();

  public static void CreateClients()
  {
    Client[] clients = Enumerable.Range (0, 1000)
     .Select (i => new Client (_host))
     .ToArray();

    // Делать что-нибудь с экземплярами класса Client...
  }
}
```

Может показаться, что после того, как метод CreateClients завершит выполнение, тысяча объектов Client станут пригодными для сборки мусора. К сожалению, на каждый объект Client имеется еще одна ссылка: объект _host, событие Click которого теперь ссылается на каждый экземпляр Client. Данный факт может остаться незамеченным, если событие Click не возникает – или если метод HostClicked не делает ничего такого, что привлекало бы внимание.

Один из способов решения проблемы – обеспечить, чтобы класс Client реализовывал интерфейс IDisposable, и в методе Dispose отсоединиться от обработчика событий:

```
public void Dispose() { _host.Click -= HostClicked; }
```

Тогда потребители класса Client освободят его экземпляры после завершения работы с ними:

```
Array.ForEach (clients, c => c.Dispose());
```

В разделе "Слабые ссылки" далее в главе мы опишем другое решение этой проблемы, которое может оказаться удобным в средах, где освобождаемые объекты, как правило, не применяются (примером такой среды может служить WPF). В действительности инфраструктура WPF предлагает класс по имени WeakEventManager, который задействует шаблон использования слабых ссылок.

В рамках WPF еще одной распространенной причиной утечек памяти является *привязка данных*.

## Таймеры

Забытые таймеры также приводят к утечкам памяти (таймеры обсуждаются в главе 22). В зависимости от вида таймера существуют два отличающихся сценария. Давайте сначала рассмотрим таймер в пространстве имен System.Timers. В следующем примере класс Foo (когда создан его экземпляр) вызывает метод tmr_Elapsed каждую секунду:

```
using System.Timers;

class Foo
{
  Timer _timer;

  Foo()
  {
    _timer = new System.Timers.Timer { Interval = 1000 };
    _timer.Elapsed += tmr_Elapsed;
    _timer.Start();
  }

  void tmr_Elapsed (object sender, ElapsedEventArgs e) { ... }
}
```

К сожалению, экземпляры Foo никогда не смогут быть обработаны сборщиком мусора! Проблема в том, что сама среда .NET Framework удерживает ссылки на активные таймеры, чтобы они могли запускать свои события Elapsed. Таким образом:

- среда .NET Framework будет удерживать _timer в активном состоянии;
- _timer будет удерживать экземпляр Foo в активном состоянии через обработчик событий tmr_Elapsed.

Решение станет очевидным, как только вы осознаете, что класс Timer реализует интерфейс IDisposable. Освобождение таймера останавливает его и гарантирует, что .NET Framework больше не ссылается на данный объект:

```
class Foo : IDisposable
{
  ...
  public void Dispose() { _timer.Dispose(); }
}
```

Полезный руководящий принцип предусматривает реализацию интерфейса IDisposable, если хоть одному полю в классе присваивается объект, который реализует IDisposable.

Касательно того, что уже обсуждалось: таймеры WPF и Windows Forms ведут себя аналогично.

Однако таймер из пространства имен System.Threading является особым. Среда .NET Framework не хранит ссылки на активные потоковые таймеры; взамен она напрямую ссылается на делегаты обратного вызова. Это значит, что если вы забудете освободить потоковый таймер, то может запуститься финализатор, который остановит и освободит таймер автоматически. Например:

```
static void Main()
{
  var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000);
  GC.Collect();
  System.Threading.Thread.Sleep (10000);  // Ждать 10 секунд
}

static void TimerTick (object notUsed) { Console.WriteLine ("tick"); }
```

Если данный пример скомпилирован в режиме выпуска (отладка отключена, а оптимизация включена), тогда таймер будет обработан сборщиком мусора и финализирован еще до того, как у него появится шанс запуститься хотя бы раз! И снова мы можем исправить положение, освободив таймер по завершении работы с ним:

```
using (var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000))
{
  GC.Collect();
  System.Threading.Thread.Sleep (10000);  // Ждать 10 секунд
}
```

Явный вызов метода tmr.Dispose в конце блока using гарантирует, что переменная tmr "используется" и потому не рассматривается сборщиком мусора как неактивная вплоть до конца блока. По иронии судьбы этот вызов метода Dispose на самом деле приводит к тому, что объект сохраняется активным *дольше*!

## Диагностика утечек памяти

Простейший способ избежать утечек управляемой памяти предполагает проведение упреждающего мониторинга потребления памяти после того, как приложение написано. Получить данные по текущему использованию памяти объектами программы можно следующим образом (аргумент true сообщает сборщику мусора о необходимости выполнения сначала процесса сборки):

```
long memoryUsed = GC.GetTotalMemory (true);
```

Если вы практикуете разработку через тесты, то есть возможность применить модульные тесты для утверждения, что память восстановлена должным образом. Если такое утверждение терпит неудачу, тогда придется проверить только изменения, которые были внесены недавно.

Находить утечки управляемой памяти в крупных приложениях помогает инструмент `windbg.exe`. Доступны также средства с дружественным графическим пользовательским интерфейсом наподобие Microsoft CLR Profiler, SciTech Memory Profiler и Red Gate ANTS Memory Profiler.

Среда CLR также открывает доступ к многочисленным счетчикам Windows WMI для помощи в мониторинге потребления ресурсов.

# Слабые ссылки

Иногда удобно удерживать ссылку на объект, который является "невидимым" сборщику мусора, в том смысле, что объект сохраняется в активном состоянии. Это называется *слабой ссылкой* и реализовано классом `System.WeakReference`.

Для использования класса `WeakReference` необходимо сконструировать его экземпляр с целевым объектом, как показано ниже:

```
var sb = new StringBuilder ("this is a test");
var weak = new WeakReference (sb);
Console.WriteLine (weak.Target);  // Выводит this is a test
```

Если на целевой объект имеется *только* одна или более слабых ссылок, то сборщик мусора считает его пригодным для сборки. После того, как целевой объект обработан сборщиком мусора, свойство `Target` экземпляра `WeakReference` получает значение `null`:

```
var weak = new WeakReference (new StringBuilder ("weak"));
Console.WriteLine (weak.Target);  // Выводит weak
GC.Collect();
Console.WriteLine (weak.Target);  // (пусто)
```

Во избежание обработки сборщиком мусора целевого объекта в промежутке между его проверкой на `null` и потреблением его следует присвоить локальной переменной:

```
var weak = new WeakReference (new StringBuilder ("weak"));
var sb = (StringBuilder) weak.Target;
if (sb != null) { /* Делать что-нибудь с sb */ }
```

Поскольку целевой объект был присвоен локальной переменной, он получил надежный корневой объект и потому не может быть обработан сборщиком мусора, пока эта переменная используется.

В приведенном ниже классе слабые ссылки применяются для отслеживания всех создаваемых объектов `Widget`, не препятствуя их обработке сборщиком мусора:

```
class Widget
{
  static List<WeakReference> _allWidgets = new List<WeakReference>();
  public readonly string Name;
  public Widget (string name)
  {
    Name = name;
    _allWidgets.Add (new WeakReference (this));
  }
```

```
  public static void ListAllWidgets()
  {
    foreach (WeakReference weak in _allWidgets)
    {
      Widget w = (Widget)weak.Target;
      if (w != null) Console.WriteLine (w.Name);
    }
  }
}
```

Единственное замечание, которое следует сделать относительно такой системы – с течением времени статический список разрастается и накапливает слабые ссылки с целевыми объектами, установленными в `null`. Таким образом, понадобится внедрить определенную стратегию очистки.

## Слабые ссылки и кеширование

Один из сценариев применения `WeakReference` связан с кешированием крупных графов объектов. Они позволяют интенсивно использующим память данным кешироваться без излишнего потребления памяти:

```
_weakCache = new WeakReference (...);  // _weakCache является полем
...
var cache = _weakCache.Target;
if (cache == null) { /* Пересоздать кеш и присвоить его _weakCache */ }
```

На практике такая стратегия может оказаться не особенно эффективной, потому что вы располагаете лишь небольшим контролем над тем, когда запускается сборщик мусора и какое поколение он выберет для проведения сборки. В частности, если ваш кеш останется в поколении Gen0, то он может быть обработан сборщиком в пределах нескольких микросекунд (не забывайте, что сборщик мусора выполняет свою работу не только тогда, когда памяти становится мало – он производит регулярную сборку и при нормальных условиях потребления памяти). В итоге, как минимум, придется организовать двухуровневый кеш, где процесс начинается с хранения сильных ссылок, которые со временем преобразуются в слабые ссылки.

## Слабые ссылки и события

Ранее уже было показано, каким образом события могут вызывать утечки управляемой памяти. Простейшее решение заключается в том, чтобы избегать подписки в таких условиях или реализовать метод `Dispose` для отмены подписки. Слабые ссылки предлагают еще одно решение.

Предположим, что есть делегат, который удерживает только слабые ссылки на свои целевые объекты. Такой делегат не будет сохранять свои целевые объекты в активном состоянии – если только не существуют независимые ссылки на них. Конечно, при этом нельзя предотвратить ситуацию, когда запущенный делегат сталкивается с висячей ссылкой на целевой объект – в период времени между моментом, когда целевой объект пригоден для сборки мусора, и моментом, когда сборщик мусора подхватит его. Чтобы такое решение было эффективным, код должен быть надежным в указанном сценарии. С учетом этого случая класс *слабого делегата* может быть реализован так, как показано ниже:

```
public class WeakDelegate<TDelegate> where TDelegate : class
{
  class MethodTarget
```

```
{
  public readonly WeakReference Reference;
  public readonly MethodInfo Method;

  public MethodTarget (Delegate d)
  {
    // d.Target будет null для целей в виде статических методов:
    if (d.Target != null) Reference = new WeakReference (d.Target);
    Method = d.Method;
  }
}
List<MethodTarget> _targets = new List<MethodTarget>();
public WeakDelegate()
{
  if (!typeof (TDelegate).IsSubclassOf (typeof (Delegate)))
    throw new InvalidOperationException
      ("TDelegate must be a delegate type");
     // TDelegate должен быть типом делегата
}

public void Combine (TDelegate target)
{
  if (target == null) return;
  foreach (Delegate d in (target as Delegate).GetInvocationList())
    _targets.Add (new MethodTarget (d));
}

public void Remove (TDelegate target)
{
  if (target == null) return;
  foreach (Delegate d in (target as Delegate).GetInvocationList())
  {
    MethodTarget mt = _targets.Find (w =>
      Equals (d.Target, w.Reference?.Target) &&
      Equals (d.Method.MethodHandle, w.Method.MethodHandle));

    if (mt != null) _targets.Remove (mt);
  }
}

public TDelegate Target
{
  get
  {
    Delegate combinedTarget = null;

    foreach (MethodTarget mt in _targets.ToArray())
    {
      WeakReference wr = mt.Reference;

      // Статический целевой объект или активный целевой объект экземпляра
      if (wr == null || wr.Target != null)
      {
        var newDelegate = Delegate.CreateDelegate (
          typeof(TDelegate), wr?.Target, mt.Method);
          combinedTarget = Delegate.Combine (combinedTarget, newDelegate);
      }
      else
        _targets.Remove (mt);
    }
```

```
      return combinedTarget as TDelegate;
    }
    set
    {
      _targets.Clear();
      Combine (value);
    }
  }
}
```

В приведенном коде демонстрируется несколько интересных моментов, связанных с C# и CLR. Для начала обратите внимание на проверку `TDelegate` на принадлежность к типу делегата в конструкторе. Это объясняется особенностью C# – следующее ограничение типа является недопустимым, т.к. C# считает `System.Delegate` специальным типом, для которого ограничения не поддерживаются:

```
... where TDelegate : Delegate  // Компилятор не разрешает поступать
                                // подобным образом
```

Взамен мы должны выбрать ограничение класса и предусмотреть в конструкторе проверку во время выполнения.

В методах `Combine` и `Remove` мы осуществляем ссылочное преобразование `target` в `Delegate` с помощью операции `as`, а не более привычной операции приведения. Причина в том, что C# запрещает использовать операцию приведения с таким параметром типа, поскольку существует потенциальная неоднозначность между *специальным преобразованием* и *ссылочным преобразованием*.

Затем мы вызываем метод `GetInvocationList`, т.к. эти методы могут быть вызваны групповыми делегатами, т.е. делегатами с более чем одним методом для вызова.

В свойстве `Target` мы строим групповой делегат, комбинирующий все делегаты, на которые имеются слабые ссылки с активными целевыми объектами, удаляя оставшиеся (висячие) ссылки из списка `_targets` во избежание его разрастания до бесконечности. (Мы могли бы усовершенствовать наш класс, делая то же самое в методе `Combine`; еще одним улучшением было бы добавление блокировок для обеспечения безопасности в отношении потоков (глава 22).) Мы также разрешаем иметь делегаты вообще без слабой ссылки; они представляют делегаты, целевой метод которых является статическим.

В следующем коде показано, как использовать готовый делегат при реализации события.

```
public class Foo
{
  WeakDelegate<EventHandler> _click = new WeakDelegate<EventHandler>();

  public event EventHandler Click
  {
    add { _click.Combine (value); } remove { _click.Remove (value); }
  }

  protected virtual void OnClick (EventArgs e)
    => _click.Target?.Invoke (this, e);
}
```

# 13

# Диагностика

Когда что-то пошло не так, важно иметь доступ к информации, которая поможет в диагностировании проблемы. Существенную помощь в этом оказывает интегрированная среда разработки или отладчик, но он обычно доступен только на этапе разработки. После поставки приложение обязано самостоятельно собирать и фиксировать диагностическую информацию. Для удовлетворения данного требования инфраструктура .NET Framework предлагает набор средств, которые позволяют регистрировать диагностическую информацию, следить за поведением приложений, обнаруживать ошибки времени выполнения и интегрироваться с инструментами отладки в случае их доступности.

Инфраструктура .NET Framework также позволяет принудительно выполнять *контракты кода*. Появившиеся во времена .NET Framework 4.0 контракты кода позволяют методам взаимодействовать через набор взаимных обязательств и инициировать *ранний* отказ, если обязательства нарушены.

Мы раскрываем контракты кода в отдельном дополнении, которое можно загрузить на веб-сайте издательства.

API-интерфейс контрактов кода (Code Contracts) был разработан подразделением Microsoft Research и требует отдельной загрузки. Несмотря на первоначальные обещания, технология не получила широкого распространения и в последние несколько лет в нее вносились минимальные изменения. Возможно, ее главным недостатком является отсутствие прямой поддержки со стороны языка C#. В результате получается более медленный цикл построения-запуска, потому что выходная сборка должна быть "переписана" после компиляции.

Типы, рассматриваемые в настоящей главе, определены главным образом в пространстве имен `System.Diagnostics`.

## Условная компиляция

С помощью *директив препроцессора* любой раздел кода C# можно компилировать условно. Директивы препроцессора представляют собой специальные инструкции для компилятора, которые начинаются с символа # (и в отличие от других конструкций C# должны полностью располагаться в одной строке). Логически они выполняются перед основной компиляцией (хотя на практике компилятор обрабатывает их во вре-

мя фазы лексического анализа). Директивами препроцессора для условной компиляции являются #if, #else, #endif и #elif.

Директива #if указывает компилятору на необходимость игнорирования раздела кода, если не определен специальный *символ*. Определить такой символ можно либо с помощью директивы #define, либо посредством ключа компиляции. Директива #define применяется к отдельному *файлу*, а ключ компиляции – ко всей *сборке*.

```
#define TESTMODE  // Директивы #define должны находиться в начале файла.
          // По соглашению имена символов записываются в верхнем регистре.
using System;

class Program
{
  static void Main()
  {
#if TESTMODE
    Console.WriteLine ("in test mode!");  // ВЫВОД: in test mode!
#endif
  }
}
```

Если удалить первую строку, то программа скомпилируется без оператора Console.WriteLine в исполняемом файле, как если бы он был закомментирован.

Директива #else аналогична оператору else языка C#, а директива #elif эквивалентна директиве #else, за которой следует #if. Операции ||, && и ! могут использоваться для выполнения операций *ИЛИ*, *И* и *НЕ*:

```
#if TESTMODE && !PLAYMODE  // если TESTMODE и не PLAYMODE
  ...
```

Однако помните, что вы не строите обычное выражение C#, а символы, над которыми вы оперируете, не имеют никакого отношения к *переменным* – статическим или любым другим.

Для определения символа на уровне сборки укажите при запуске компилятора ключ /define:

```
csc Program.cs /define:TESTMODE,PLAYMODE
```

Среда Visual Studio позволяет вводить символы условной компиляции в окне свойств проекта.

Если вы определили символ на уровне сборки и затем хотите отменить его определение для какого-то файла, тогда применяйте директиву #undef.

## Сравнение условной компиляции и статических переменных-флагов

Предыдущий пример можно было бы реализовать с использованием простого статического поля:

```
static internal bool TestMode = true;

static void Main()
{
  if (TestMode) Console.WriteLine ("in test mode!");
}
```

Преимущество такого подхода связано с возможностью конфигурирования во время выполнения. Итак, почему выбирают условную компиляцию? Причина в том, что

условная компиляция способна решать задачи, которые нельзя решить посредством переменных-флагов, например:

- условное включение атрибута;
- изменение типа, объявляемого для переменной;
- переключение между разными пространствами имен или псевдонимами типов в директиве `using`:

```
using TestType =
  #if V2
    MyCompany.Widgets.GadgetV2;
  #else
    MyCompany.Widgets.Gadget;
  #endif
```

Под директивой условной компиляции можно даже реализовать крупную перестройку кода, так что появится возможность немедленного переключения между старой версией и новой. Вдобавок можно писать библиотеки, которые компилируются для нескольких версий .NET Framework, используя в своих интересах самые новые функциональные средства .NET Framework, когда они доступны.

Еще одно преимущество условной компиляции связано с тем, что отладочный код может ссылаться на типы в сборках, которые не включаются при развертывании.

## Атрибут `Conditional`

Атрибут `Conditional` указывает компилятору на необходимость игнорирования любых обращений к определенному классу или методу, если заданный символ не был определен.

Чтобы посмотреть, насколько это полезно, представим, что мы реализуем метод для регистрации информации о состоянии следующим образом:

```
static void LogStatus (string msg)
{
  string logFilePath = ...
  System.IO.File.AppendAllText (logFilePath, msg + "\r\n");
}
```

Теперь предположим, что его нужно выполнять, только если определен символ `LOGGINGMODE`. Первое решение предусматривает помещение всех вызовов метода `LogStatus` внутрь директивы `#if`:

```
#if LOGGINGMODE
LogStatus ("Message Headers: " + GetMsgHeaders());
#endif
```

Результат получается идеальным, но постоянно писать такой код утомительно. Второе решение заключается в помещении директивы `#if` внутрь самого метода `LogStatus`. Однако это проблематично, поскольку `LogStatus` должен вызываться так:

```
LogStatus ("Message Headers: " + GetComplexMessageHeaders());
```

Метод `GetComplexMessageHeaders` будет вызываться всегда, что приведет к снижению производительности.

Мы можем скомбинировать функциональность первого решения с удобством второго, присоединив к методу `LogStatus` атрибут `Conditional` (который определен в пространстве имен `System.Diagnostics`):

```
[Conditional ("LOGGINGMODE")]
static void LogStatus (string msg)
{
  ...
}
```

В результате компилятор трактует вызовы `LogStatus`, как если бы они были помещены внутрь директивы `#if LOGGINGMODE`. Когда символ не определен, любые обращения к методу `LogStatus` полностью исключаются из компиляции, в том числе и выражения оценки его аргумента. (Следовательно, будут пропускаться любые выражения, дающие побочные эффекты.) Такой прием работает, даже если метод `LogStatus` и вызывающий класс находятся в разных сборках.

Еще одно преимущество конструкции `[Conditional]` в том, что условная проверка выполняется, когда компилируется *вызывающий класс*, а не *вызываемый метод*. Это удобно, т.к. позволяет написать библиотеку, которая содержит методы вроде `LogStatus`, и построить только одну версию данной библиотеки.

Во время выполнения атрибут `Conditional` игнорируется – он представляет собой исключительно инструкцию для компилятора.

### Альтернативы атрибуту `Conditional`

Атрибут `Conditional` бесполезен, когда во время выполнения необходима возможность динамического включения или отключения функциональности: вместо него должен применяться подход на основе переменных. Остается открытым вопрос о том, как элегантно обойти оценку аргументов при вызове условных методов регистрации. Проблема решается с помощью функционального подхода:

```
using System;
using System.Linq;

class Program
{
  public static bool EnableLogging;

  static void LogStatus (Func<string> message)
  {
    string logFilePath = ...
    if (EnableLogging)
      System.IO.File.AppendAllText (logFilePath, message() + "\r\n");
  }
}
```

Лямбда-выражение позволяет вызывать данный метод без разбухания синтаксиса:

```
LogStatus ( () => "Message Headers: " + GetComplexMessageHeaders() );
```

Если значение `EnableLogging` равно `false`, тогда вызов метода `GetComplexMessageHeaders` никогда не оценивается.

## Классы `Debug` и `Trace`

`Debug` и `Trace` – статические классы, которые предлагают базовые возможности регистрации и утверждений. Указанные два класса очень похожи; основное отличие

связано с тем, для чего они предназначены. Класс Debug предназначен для отладочных сборок, а класс Trace – для отладочных и окончательных сборок. Чтобы достичь таких целей:

- все методы класса Debug определены с атрибутом [Conditional("DEBUG")];
- все методы класса Trace определены с атрибутом [Conditional("TRACE")].

Это означает, что все обращения к Debug или Trace исключаются компилятором, если только не определен символ DEBUG или TRACE. По умолчанию в Visual Studio определены оба символа, DEBUG и TRACE, в конфигурации *отладки* и один лишь символ TRACE в конфигурации *выпуска*.

Классы Debug и Trace предоставляют методы Write, WriteLine и WriteIf. По умолчанию они отправляют сообщения в окно вывода отладчика:

```
Debug.Write     ("Data");
Debug.WriteLine (23 * 34);
int x = 5, y = 3;
Debug.WriteIf   (x > y, "x is greater than y");
```

Класс Trace также предлагает методы TraceInformation, TraceWarning и TraceError. Отличия в поведении между ними и методами Write зависят от активных прослушивателей TraceListener (мы рассмотрим их в разделе "TraceListener" далее в главе).

## Fail и Assert

Классы Debug и Trace предоставляют методы Fail и Assert. Метод Fail отправляет сообщение каждому экземпляру TraceListener из коллекции Listeners внутри класса Debug или Trace (как будет показано в следующем разделе), которые по умолчанию записывают переданное сообщение в вывод отладки, а также отображают его в диалоговом окне:

```
Debug.Fail ("File data.txt does not exist!"); //Файл data.txt не существует!
```

В открывшемся диалоговом окне предлагается выбрать дальнейшее действие: игнорировать, прервать или повторить. Последнее действие затем позволяет присоединить отладчик, что удобно для более точного диагностирования проблемы.

Метод Assert просто вызывает метод Fail, если аргумент типа bool равен false; это называется *созданием утверждения* и указывает на ошибку в коде, если оно нарушено. Можно также задать необязательное сообщение об ошибке:

```
Debug.Assert (File.Exists ("data.txt"), "File data.txt does not exist!");
var result = ...
Debug.Assert (result != null);
```

Методы Write, Fail и Assert также перегружены, чтобы вдобавок к сообщению принимать строковую категорию, которая может быть полезна при обработке вывода.

Альтернативой утверждению будет генерация исключения, если противоположное условие равно true. Это общепринятый подход при проверке достоверности аргументов метода:

```
public void ShowMessage (string message)
{
  if (message == null) throw new ArgumentNullException ("message");
  ...
}
```

Такие "утверждения" компилируются безусловным образом и обладают меньшей гибкостью в том, что не позволяют управлять результатом отказавшего утверждения через экземпляры `TraceListener`. К тому же формально они не являются утверждениями. Утверждение представляет собой то, что в случае нарушения говорит об ошибке в коде текущего метода. Генерация исключения на основе проверки достоверности аргумента указывает на ошибку в коде *вызывающего компонента*.

## TraceListener

Классы `Debug` и `Trace` имеют свойство `Listeners`, которое является статической коллекцией экземпляров `TraceListener`. Они отвечают за обработку содержимого, выдаваемого методами `Write`, `Fail` и `Trace`.

По умолчанию коллекция `Listeners` в обоих классах включает единственный прослушиватель (`DefaultTraceListener`). Стандартный прослушиватель обладает двумя основными возможностями.

- В случае подключения к отладчику наподобие встроенного в Visual Studio сообщения записываются в окно вывода отладки; иначе содержимое сообщения игнорируется.
- Когда вызывается метод `Fail` (или утверждение не выполняется), отображается диалоговое окно, предлагающее пользователю выбрать дальнейшее действие – продолжить, прервать или повторить (присоединение/отладку) – независимо от того, подключен ли отладчик.

Вы можете изменить такое поведение, (необязательно) удалив стандартный прослушиватель и затем добавив один или большее число собственных прослушивателей. Прослушиватели трассировки можно написать с нуля (создавая подкласс класса `TraceListener`) или воспользоваться одним из предопределенных типов:

- `TextWriterTraceListener` записывает в `Stream` или `TextWriter` либо добавляет в файл;
- `EventLogTraceListener` записывает в журнал событий Windows;
- `EventProviderTraceListener` записывает в подсистему трассировки событий для Windows (Event Tracing for Windows – ETW) в Windows Vista и последующих версиях;
- `WebPageTraceListener` записывает на веб-страницу ASP.NET.

Класс `TextWriterTraceListener` имеет подклассы `ConsoleTraceListener`, `DelimitedListTraceListener`, `XmlWriterTraceListener` и `EventSchemaTraceListener`.

Ни один из перечисленных выше прослушивателей не отображает диалоговое окно, когда вызывается метод `Fail` – таким поведением обладает только класс `DefaultTraceListener`.

В следующем примере очищается стандартный прослушиватель `Trace`, после чего добавляются три прослушивателя – первый дописывает в файл, второй выводит на консоль и третий записывает в журнал событий Windows:

```
// Очистить стандартный прослушиватель:
Trace.Listeners.Clear();

// Добавить средство записи, дописывающее в файл trace.txt:
Trace.Listeners.Add (new TextWriterTraceListener ("trace.txt"));
```

```
// Получить выходной поток Console и добавить его в качестве прослушивателя:
System.IO.TextWriter tw = Console.Out;
Trace.Listeners.Add (new TextWriterTraceListener (tw));

// Настроить исходный файл журнала событий и создать/добавить прослушиватель.
// Метод CreateEventSource требует повышения полномочий до уровня
// администратора, так что это обычно будет делаться при установке приложения.
if (!EventLog.SourceExists ("DemoApp"))
  EventLog.CreateEventSource ("DemoApp", "Application");

Trace.Listeners.Add (new EventLogTraceListener ("DemoApp"));
```

(Добавлять прослушиватели можно также через файл конфигурации приложения; такой подход удобен тем, что предоставляет тестировщикам возможность конфигурировать трассировку после сборки приложения — см. статью MSDN по адресу `https://msdn.microsoft.com/ru-ru/library/sk36c28t.aspx`.)

В случае журнала событий Windows сообщения, записываемые с помощью метода `Write`, `Fail` или `Assert`, всегда отображаются в программе "Просмотр событий" как сообщения уровня сведений. Однако сообщения, которые записываются посредством методов `TraceWarning` и `TraceError`, отображаются как предупреждения или ошибки.

Класс `TraceListener` также имеет свойство `Filter` типа `TraceFilter`, которое можно устанавливать для управления тем, будет ли сообщение записано данным прослушивателем. Чтобы сделать это, нужно либо создать экземпляр одного из предопределенных подклассов (`EventTypeFilter` или `SourceFilter`), либо создать подкласс класса `TraceFilter` и переопределить метод `ShouldTrace`. Подобный прием можно использовать, скажем, для фильтрации по категории.

В классе `TraceListener` также определены свойства `IndentLevel` и `IndentSize` для управления отступами и свойство `TraceOutputOptions` для записи дополнительных данных:

```
TextWriterTraceListener tl = new TextWriterTraceListener (Console.Out);
tl.TraceOutputOptions = TraceOptions.DateTime | TraceOptions.Callstack;
```

Свойство `TraceOutputOptions` применяется при использовании методов `Trace`:

```
Trace.TraceWarning ("Orange alert");
```

```
DiagTest.vshost.exe Warning: 0 : Orange alert
    DateTime=2018-03-08T05:57:13.6250000Z
    Callstack=   at System.Environment.GetStackTrace(Exception e, Boolean
needFileInfo)
    at System.Environment.get_StackTrace()    at ...
```

## Сброс и закрытие прослушивателей

Некоторые прослушиватели, такие как `TextWriterTraceListener`, в итоге производят запись в поток, подлежащий кешированию. Результатом будут два последствия.

- Сообщение может не появиться в выходном потоке или файле немедленно.
- Перед завершением приложения прослушиватель потребуется закрыть (или, по крайней мере, сбросить); в противном случае потеряется все то, что находится в кеше (по умолчанию до 4 Кбайт данных, если осуществляется запись в файл).

Классы `Trace` и `Debug` предлагают статические методы `Close` и `Flush`, которые вызывают `Close` или `Flush` на всех прослушивателях (а эти методы в свою очередь вызывают `Close` или `Flush` на любых лежащих в основе средствах записи и потоках).

Метод `Close` неявно вызывает `Flush`, закрывает файловые дескрипторы и предотвращает дальнейшую запись данных.

В качестве общего правила: метод `Close` должен вызываться перед завершением приложения, а метод `Flush` – каждый раз, когда нужно удостовериться, что текущие данные сообщений записаны. Такое правило применяется при использовании прослушивателей, основанных на потоках или файлах.

Классы `Trace` и `Debug` также предоставляют свойство `AutoFlush`, которое в случае равенства `true` приводит к вызову метода `Flush` после каждого сообщения.

Если применяются прослушиватели, основанные на потоках или файлах, то эффективной политикой будет установка в `true` свойства `AutoFlush` для экземпляров `Debug` и `Trace`. Иначе при возникновении исключения или критической ошибки последние 4 Кбайт диагностической информации могут быть утеряны.

# Интеграция с отладчиком

Иногда для приложения удобно взаимодействовать с каким-нибудь отладчиком, если он доступен. На этапе разработки отладчик обычно предоставляется IDE-средой (например, Visual Studio), а после развертывания отладчиком, скорее всего, будет:

- DbgCLR;
- один из низкоуровневых инструментов отладки, такой как WinDbg, Cordbg или Mdbg.

Инструмент DbgCLR является усеченной версией Visual Studio, в которой оставлен только отладчик, и он свободно загружается в составе .NET Framework SDK. Это простейший вариант отладки при отсутствии доступа к IDE-среде, хотя он требует загрузки полного SDK.

## Присоединение и останов

Статический класс `Debugger` из пространства имен `System.Diagnostics` предлагает базовые функции для взаимодействия с отладчиком, а именно – `Break`, `Launch`, `Log` и `IsAttached`.

Для отладки к приложению сначала потребуется присоединить отладчик. В случае запуска приложения из IDE-среды отладчик присоединяется автоматически, если только не запрошено противоположное (выбором пункта меню Start without debugging (Запустить без отладки)). Однако иногда запускать приложение в режиме отладки внутри IDE-среды неудобно или невозможно. Примером может быть приложение Windows-службы или (по иронии судьбы) визуальный редактор Visual Studio. Одно из решений предполагает запуск приложения обычным образом с последующим выбором пункта меню Debug Process (Отладить процесс) в IDE-среде. Тем не менее, при таком подходе нет возможности поместить точку останова в самое начало процесса выполнения программы.

Обходной путь предусматривает вызов метода `Debugger.Break` внутри приложения. Данный метод запускает отладчик, присоединяется к нему и приостанавливает выполнение в точке вызова. (Метод `Launch` делает то же самое, но не приостанавливает выполнение.) После присоединения с помощью метода `Log` сообщения можно отправлять прямо в окно вывода отладчика. Состояние присоединения к отладчику выясняется через свойство `IsAttached`.

## Атрибуты отладчика

Атрибуты `DebuggerStepThrough` и `DebuggerHidden` предоставляют указания отладчику о том, как обрабатывать пошаговое выполнение для конкретного метода, конструктора или класса.

Атрибут `DebuggerStepThrough` требует, чтобы отладчик прошел через функцию без взаимодействия с пользователем. Данный атрибут полезен для автоматически сгенерированных методов и прокси-методов, которые перекладывают выполнение реальной работы на какие-то другие методы. В последнем случае отладчик будет отображать прокси-метод в стеке вызовов, даже когда точка останова находится внутри "реального" метода – если только не добавить также атрибут `DebuggerHidden`. Упомянутые атрибуты можно комбинировать на прокси-методах, чтобы помочь пользователю сосредоточить внимание на отладке прикладной логики, а не связующего вспомогательного кода:

```
[DebuggerStepThrough, DebuggerHidden]
void DoWorkProxy()
{
  // Настройка...
  DoWork();
  // Освобождение...
}
void DoWork() {...}  // Реальный метод...
```

# Процессы и потоки процессов

В последнем разделе главы 6 приводились объяснения, как запустить новый процесс с помощью метода `Process.Start`. Класс `Process` также позволяет запрашивать и взаимодействовать с другими процессами, выполняющимися на том же самом или другом компьютере. Класс `Process` является частью .NET Standard 2.0, хотя для платформы UWP его возможности ограничены.

## Исследование выполняющихся процессов

Методы `Process.GetProcess`*XXX* извлекают специфический процесс по имени либо идентификатору или все процессы, выполняющиеся на текущей либо указанной машине. Сюда входят как управляемые, так и неуправляемые процессы. Каждый экземпляр `Process` имеет множество свойств, отражающих статистические сведения, такие как имя, идентификатор, приоритет, утилизация памяти и процессора, оконные дескрипторы и т.д. В следующем примере производится перечисление всех процессов, функционирующих на текущем компьютере:

```
foreach (Process p in Process.GetProcesses())
using (p)
{
  Console.WriteLine (p.ProcessName);
  Console.WriteLine ("   PID:      " + p.Id);      // Идентификатор процесса
  Console.WriteLine ("   Memory:   " + p.WorkingSet64); // Память
  Console.WriteLine ("   Threads:  " + p.Threads.Count); //Количество потоков
}
```

Метод `Process.GetCurrentProcess` возвращает текущий процесс. Если были созданы дополнительные домены приложений, то все они будут разделять один и тот же процесс.

Завершить процесс можно вызовом его метода Kill.

## Исследование потоков в процессе

С помощью свойства Process.Threads можно также реализовать перечисление потоков других процессов. Однако вместо объектов System.Threading.Thread будут получены объекты ProcessThread, которые предназначены для решения административных задач, а не задач, касающихся синхронизации. Объект ProcessThread предоставляет диагностическую информацию о лежащем в основе потоке и позволяет управлять некоторыми связанными с ним аспектами, такими как приоритет и родство:

```
public void EnumerateThreads (Process p)
{
  foreach (ProcessThread pt in p.Threads)
  {
    Console.WriteLine (pt.Id);
    Console.WriteLine ("   State:    " + pt.ThreadState);        // Состояние
    Console.WriteLine ("   Priority: " + pt.PriorityLevel);     // Приоритет
    Console.WriteLine ("   Started:  " + pt.StartTime);         // Запущен
    Console.WriteLine ("   CPU time: " + pt.TotalProcessorTime); // Время ЦП
  }
}
```

# StackTrace и StackFrame

Классы StackTrace и StackFrame предлагают допускающее только чтение представление стека вызовов и являются частью стандартной инфраструктуры .NET Framework для настольных приложений. Трассировки стека можно получать для текущего потока, другого потока в том же самом процессе или объекта Exception. Такая информация полезна в основном для диагностических целей, хотя ее также можно использовать и в программировании. Экземпляр StackTrace представляет полный стек вызовов, а StackFrame — одиночный вызов метода внутри стека.

Если экземпляр StackTrace создается без аргументов (или с аргументом типа bool), тогда будет получен снимок стека вызовов текущего потока. Когда аргумент типа bool равен true, он инструктирует StackTrace о необходимости чтения файлов *.pdb* (project debug — отладка проекта) сборки, если они существуют, предоставляя доступ к данным об именах файлов, номерах строк и позициях в строках. Файлы отладки проекта генерируются в случае компиляции с ключом /debug. (Среда Visual Studio компилирует с этим ключом, если не затребовано построение окончательной сборки через дополнительные параметры построения (в диалоговом окне Advanced Build Settings).)

После получения экземпляра StackTrace можно исследовать любой отдельный фрейм с помощью вызова метода GetFrame или же все фреймы посредством вызова GetFrames:

```
static void Main() { A (); }
static void A()    { B (); }
static void B()    { C (); }
static void C()
{
  StackTrace s = new StackTrace (true);
  Console.WriteLine ("Total frames:   " + s.FrameCount);     // Всего фреймов
```

```
  Console.WriteLine ("Current method: " + s.GetFrame(0).GetMethod().Name);
                    // Текущий метод
  Console.WriteLine ("Calling method: " + s.GetFrame(1).GetMethod().Name);
                    // Вызывающий метод
  Console.WriteLine ("Entry method:   " + s.GetFrame
                    // Входной метод
                         (s.FrameCount-1).GetMethod().Name);
  Console.WriteLine ("Call Stack:");
                    // Стек вызовов
  foreach (StackFrame f in s.GetFrames())
    Console.WriteLine (
      "  File: "   + f.GetFileName() +                  /* Файл     */
      "  Line: "   + f.GetFileLineNumber() +            /* Строка   */
      "  Col: "    + f.GetFileColumnNumber() +          /* Колонка  */
      "  Offset: " + f.GetILOffset() +                  /* Смещение */
      "  Method: " + f.GetMethod().Name);               /* Метод    */
}
```

Ниже показан вывод:

```
Total frames:   4
Current method: C
Calling method: B
Entry method: Main
Call stack:
  File: C:\Test\Program.cs  Line: 15  Col: 4  Offset: 7  Method: C
  File: C:\Test\Program.cs  Line: 12  Col: 22  Offset: 6  Method: B
  File: C:\Test\Program.cs  Line: 11  Col: 22  Offset: 6  Method: A
  File: C:\Test\Program.cs  Line: 10  Col: 25  Offset: 6  Method: Main
```

Смещение IL указывает смещение инструкции, которая будет выполнена *следующей*, а не той, что выполняется в текущий момент. Тем не менее, номера строк и колонок (при наличии файла `.pdb`) обычно отражают действительную точку выполнения.

Так происходит оттого, что среда CLR делает все возможное для *выведения* фактической точки выполнения при вычислении строки и колонки из смещения IL. Компилятор генерирует код IL так, чтобы сделать это реальным, при необходимости вставляя в поток IL инструкции `nop` (no-operation – нет операции).

Однако компиляция с включенной оптимизацией запрещает вставку инструкций `nop`, а потому трассировка стека может отражать номера строки и колонки, где расположен оператор, который будет выполняться следующим. Получение удобной трассировки стека еще более затрудняется тем фактом, что оптимизация может быть связана и с применением других трюков, таких как устранение целых методов.

Сокращенный способ получения важной информации для полного экземпляра `StackTrace` предусматривает вызов на нем метода `ToString`. Вот как могут выглядеть результаты:

```
at DebugTest.Program.C() in C:\Test\Program.cs:line 16
at DebugTest.Program.B() in C:\Test\Program.cs:line 12
at DebugTest.Program.A() in C:\Test\Program.cs:line 11
at DebugTest.Program.Main() in C:\Test\Program.cs:line 10
```

Чтобы получить трассировку стека для другого потока, конструктору `StackTrace` необходимо передать другой экземпляр `Thread`. Такой прием может оказаться удобной стратегией для профилирования программы, хотя на время получения трассировки стека поток должен быть приостановлен. В действительности это сделать достаточно сложно, не рискуя войти в состояние взаимоблокировки — мы продемонстрируем надежный подход в разделе "`Suspend` и `Resume`" главы 22.

Трассировку стека можно также получить для объекта `Exception`, передав его конструктору `StackTrace` (она покажет, что именно привело к генерации исключения).

Класс `Exception` уже имеет свойство `StackTrace`; тем не менее, оно возвращает простую строку, а не объект `StackTrace`. Объект `StackTrace` намного более полезен при регистрации исключений, возникающих после развертывания (когда файлы `.pdb` уже не доступны), поскольку вместо номеров строк и колонок в журнале можно регистрировать *смещение IL*. С помощью смещения IL и утилиты `ildasm` несложно выяснить, внутри какого метода возникла ошибка.

# Журналы событий Windows

Платформа Win32 предоставляет централизованный механизм регистрации в форме журналов событий Windows.

Применяемые ранее классы `Debug` и `Trace` осуществляли запись в журнал событий Windows, если был зарегистрирован прослушиватель `EventLogTraceListener`. Тем не менее, посредством класса `EventLog` можно записывать напрямую в журнал событий Windows, не используя классы `Trace` или `Debug`. Класс `EventLog` можно также применять для чтения и мониторинга данных, связанных с событиями.

Выполнять запись в журнал событий Windows имеет смысл в приложении Windows-службы, поскольку если что-то идет не так, то нет никакой возможности отобразить пользовательский интерфейс, который направил бы пользователя на специфический файл, куда была занесена диагностическая информация. Кроме того, запись в журнал событий Windows является общепринятой практикой для служб, так что данный журнал будет первым местом, где администратор начнет выяснять причины отказа той или иной службы.

Класс `EventLog` не является частью .NET Standard и он не доступен для приложений UWP и .NET Core.

Существуют три стандартных журнала событий Windows со следующими именами:

- `Application` (приложение)
- `System` (система)
- `Security` (безопасность)

Большинство приложений обычно производят запись в журнал *`Application`*.

## Запись в журнал событий

Ниже перечислены шаги, которые понадобится выполнить для записи в журнал событий Windows.

1. Выберите один из трех журналов событий (обычно Application).
2. Примите решение относительно *имени источника* и при необходимости создайте его.
3. Вызовите метод EventLog.WriteEntry с именем журнала, именем источника и данными сообщения.

*Имя источника* – это просто идентифицируемое имя вашего приложения. Перед использованием имя источника должно быть зарегистрировано; такую функцию выполняет метод CreateEventSource. Затем можно вызывать метод WriteEntry:

```
const string SourceName = "MyCompany.WidgetServer";

// Метод CreateEventSource требует наличия административных полномочий,
// поэтому данный код обычно выполняется при установке приложения.
if (!EventLog.SourceExists (SourceName))
  EventLog.CreateEventSource (SourceName, "Application");

EventLog.WriteEntry (SourceName,
  "Service started; using configuration file=...",
  EventLogEntryType.Information);
```

Перечисление EventLogEntryType содержит следующие значения: Information, Warning, Error, SuccessAudit и FailureAudit. Каждое значение обеспечивает отображение разного значка в программе просмотра событий Windows. Можно также указать необязательные категорию и идентификатор события (произвольные числа по вашему выбору) и предоставить дополнительные двоичные данные.

Метод CreateEventSource также позволяет задавать имя машины, что приведет к записи в журнал событий на другом компьютере при наличии достаточных полномочий.

## Чтение журнала событий

Для чтения журнала событий необходимо создать экземпляр класса EventLog с именем нужного журнала и дополнительно именем компьютера, если журнал находится на другом компьютере. Впоследствии любая запись журнала может быть прочитана с помощью свойства Entries типа коллекции:

```
EventLog log = new EventLog ("Application");

Console.WriteLine ("Total entries: " + log.Entries.Count);  // Всего записей

EventLogEntry last = log.Entries [log.Entries.Count - 1];
Console.WriteLine ("Index:    " + last.Index);            // Индекс
Console.WriteLine ("Source:   " + last.Source);           // Источник
Console.WriteLine ("Type:     " + last.EntryType);        // Тип
Console.WriteLine ("Time:     " + last.TimeWritten);      // Время
Console.WriteLine ("Message:  " + last.Message);          // Сообщение
```

С помощью статического метода EventLog.GetEventLogs можно перечислить все журналы на текущем (или другом) компьютере (действие требует наличия административных привилегий):

```
foreach (EventLog log in EventLog.GetEventLogs())
  Console.WriteLine (log.LogDisplayName);
```

Обычно данный код выводит минимум Application, Security и System.

## Мониторинг журнала событий

Организовать оповещение о появлении любой записи в журнале событий Windows можно посредством события `EntryWritten`. Прием работает для журналов событий на локальном компьютере независимо от того, какое приложение записало событие.

Чтобы включить мониторинг журнала событий, необходимо выполнить следующие действия.

1. Создайте экземпляр `EventLog` и установите его свойство `EnableRaisingEvents` в `true`.
2. Обработайте событие `EntryWritten`.

Вот пример:

```
static void Main()
{
  using (var log = new EventLog ("Application"))
  {
    log.EnableRaisingEvents = true;
    log.EntryWritten += DisplayEntry;
    Console.ReadLine();
  }
}

static void DisplayEntry (object sender, EntryWrittenEventArgs e)
{
  EventLogEntry entry = e.Entry;
  Console.WriteLine (entry.Message);
}
```

# Счетчики производительности

Обсуждаемые ранее механизмы регистрации удобны для накопления информации, которая будет анализироваться в будущем. Однако чтобы получить представление о текущем состоянии приложения (или системы в целом), необходим какой-то подход реального времени. Решением такой потребности в Win32 является инфраструктура для мониторинга производительности, которая состоит из набора счетчиков производительности, открываемых системой и приложениями, и оснасток консоли управления Microsoft (Microsoft Management Console – MMC), используемых для отслеживания этих счетчиков в реальном времени.

Счетчики производительности сгруппированы в категории, такие как "Система", "Процессор", "Память .NET CLR" и т.д. В инструментах с графическим пользовательским интерфейсом такие категории иногда называются "объектами производительности". Каждая категория группирует связанный набор счетчиков производительности, отслеживающих один аспект системы или приложения. Примерами счетчиков производительности в категории "Память .NET CLR" могут быть "% времени сборки мусора", "# байтов во всех кучах" и "Выделено байтов/с".

Каждая категория может дополнительно иметь один или более экземпляров, допускающих независимый мониторинг. Это полезно, например, для счетчика производительности "% процессорного времени" из категории "Процессор", который позволяет отслеживать утилизацию центрального процессора. На многопроцессорной машине данный счетчик поддерживает экземпляры для всех процессоров, позволяя независимо проводить мониторинг использования каждого процессора.

В последующих разделах будет показано, как решать часто встречающиеся задачи, такие как определение открытых счетчиков, отслеживание счетчиков и создание собственных счетчиков для отображения информации о состоянии приложения.

Чтение счетчиков производительности или категорий может требовать наличия административных полномочий на локальном или целевом компьютере в зависимости от того, к чему производится доступ.

## Перечисление доступных счетчиков производительности

В следующем примере осуществляется перечисление всех доступных счетчиков производительности на компьютере. В случае если счетчик имеет экземпляры, тогда перечисляются счетчики для каждого экземпляра:

```
PerformanceCounterCategory[] cats =
  PerformanceCounterCategory.GetCategories();

foreach (PerformanceCounterCategory cat in cats)
{
  Console.WriteLine ("Category: " + cat.CategoryName);     // Категория

  string[] instances = cat.GetInstanceNames();
  if (instances.Length == 0)
  {
    foreach (PerformanceCounter ctr in cat.GetCounters())
      Console.WriteLine ("  Counter: " + ctr.CounterName); // Счетчик
  }
  else   // Вывести счетчики, имеющие экземпляры
  {
    foreach (string instance in instances)
    {
      Console.WriteLine ("  Instance: " + instance);        // Экземпляр
      if (cat.InstanceExists (instance))
        foreach (PerformanceCounter ctr in cat.GetCounters (instance))
          Console.WriteLine ("    Counter: " + ctr.CounterName);  // Счетчик
    }
  }
}
```

Результат содержит свыше 10 000 строк! Его получение также занимает некоторое время, поскольку реализация метода `PerformanceCounterCategory.InstanceExists` неэффективна. В реальной системе настолько детальная информация извлекается только по требованию.

В приведенном далее примере с помощью запроса LINQ извлекаются лишь счетчики производительности, связанные с .NET, а результат помещается в XML-файл:

```
var x =
  new XElement ("counters",
    from PerformanceCounterCategory cat in
        PerformanceCounterCategory.GetCategories()
    where cat.CategoryName.StartsWith (".NET")
    let instances = cat.GetInstanceNames()
    select new XElement ("category",
      new XAttribute ("name", cat.CategoryName),
      instances.Length == 0
```

```
      ?
        from c in cat.GetCounters()
        select new XElement ("counter",
          new XAttribute ("name", c.CounterName))
      :
        from i in instances
        select new XElement ("instance", new XAttribute ("name", i),
          !cat.InstanceExists (i)
          ?
            null
          :
            from c in cat.GetCounters (i)
            select new XElement ("counter",
              new XAttribute ("name", c.CounterName))
        )
    )
  );
x.Save ("counters.xml");
```

## Чтение данных счетчика производительности

Чтобы извлечь значение счетчика производительности, необходимо создать объект `PerformanceCounter` и затем вызвать его метод `NextValue` или `NextSample`. Метод `NextValue` возвращает простое значение типа `float`, а метод `NextSample` – объект `CounterSample`, который открывает доступ к более широкому набору свойств наподобие `CounterFrequency`, `TimeStamp`, `BaseValue` и `RawValue`.

Конструктор `PerformanceCounter` принимает имя категории, имя счетчика и необязательный экземпляр. Таким образом, чтобы отобразить сведения о текущей утилизации всех процессоров, потребуется написать следующий код:

```
using (PerformanceCounter pc = new PerformanceCounter ("Processor",
                                                        "% Processor Time",
                                                        "_Total"))
  Console.WriteLine (pc.NextValue());
```

А вот как отобразить данные по потреблению "реальной" (т.е. закрытой) памяти текущим процессом:

```
string procName = Process.GetCurrentProcess().ProcessName;
using (PerformanceCounter pc = new PerformanceCounter ("Process",
                                                        "Private Bytes",
                                                         procName))
  Console.WriteLine (pc.NextValue());ы
```

Класс `PerformanceCounter` не открывает доступ к событию `ValueChanged`, поэтому для отслеживания изменений потребуется реализовать опрос. В следующем примере опрос производится каждые 200 миллисекунд – пока не поступит сигнал завершения от `EventWaitHandle`:

```
// Необходимо импортировать пространства имен System.Threading и System.Diagnostics
static void Monitor (string category, string counter, string instance,
                     EventWaitHandle stopper)
{
  if (!PerformanceCounterCategory.Exists (category))
    throw new InvalidOperationException ("Category does not exist");
                                        // Категория не существует
```

```
  if (!PerformanceCounterCategory.CounterExists (counter, category))
    throw new InvalidOperationException ("Counter does not exist");
                                        // Счетчик не существует
  if (instance == null) instance = ""; //"" == экземпляры отсутствуют (не null!)
  if (instance != "" &&
     !PerformanceCounterCategory.InstanceExists (instance, category))
    throw new InvalidOperationException ("Instance does not exist");
                                        // Экземпляр не существует
  float lastValue = 0f;
  using (PerformanceCounter pc = new PerformanceCounter (category,
                                                         counter, instance))
    while (!stopper.WaitOne (200, false))
    {
      float value = pc.NextValue();
      if (value != lastValue)  // Записывать значение, только
      {                        // если оно изменилось.
        Console.WriteLine (value);
        lastValue = value;
      }
    }
}
```

Ниже показано, как применять метод `Monitor` для одновременного мониторинга работы процессора и жесткого диска:

```
static void Main()
{
  EventWaitHandle stopper = new ManualResetEvent (false);

  new Thread (() =>
    Monitor ("Processor", "% Processor Time", "_Total", stopper)
  ).Start();

  new Thread (() =>
    Monitor ("LogicalDisk", "% Idle Time", "C:", stopper)
  ).Start();

  // Проведение мониторинга; для завершения нужно нажать любую клавишу
  Console.WriteLine ("Monitoring - press any key to quit");
  Console.ReadKey();
  stopper.Set();
}
```

## Создание счетчиков и запись данных о производительности

Перед записью данных счетчика производительности понадобится создать категорию производительности и счетчик. Категория производительности должна быть создана наряду со всеми принадлежащими ей счетчиками за один шаг:

```
string category = "Nutshell Monitoring";

// Мы создадим два счетчика в следующей категории:
string eatenPerMin = "Macadamias eaten so far";
string tooHard = "Macadamias deemed too hard";

if (!PerformanceCounterCategory.Exists (category))
{
  CounterCreationDataCollection cd = new CounterCreationDataCollection();
  cd.Add (new CounterCreationData (eatenPerMin,
```

```
            "Number of macadamias consumed, including shelling time",
            PerformanceCounterType.NumberOfItems32));
        cd.Add (new CounterCreationData (tooHard,
            "Number of macadamias that will not crack, despite much effort",
            PerformanceCounterType.NumberOfItems32));
        PerformanceCounterCategory.Create (category, "Test Category",
          PerformanceCounterCategoryType.SingleInstance, cd);
    }
```

Новые счетчики появятся в инструменте мониторинга производительности Windows при выборе опции Add Counters (Добавить счетчики), как показано на рис. 13.1.

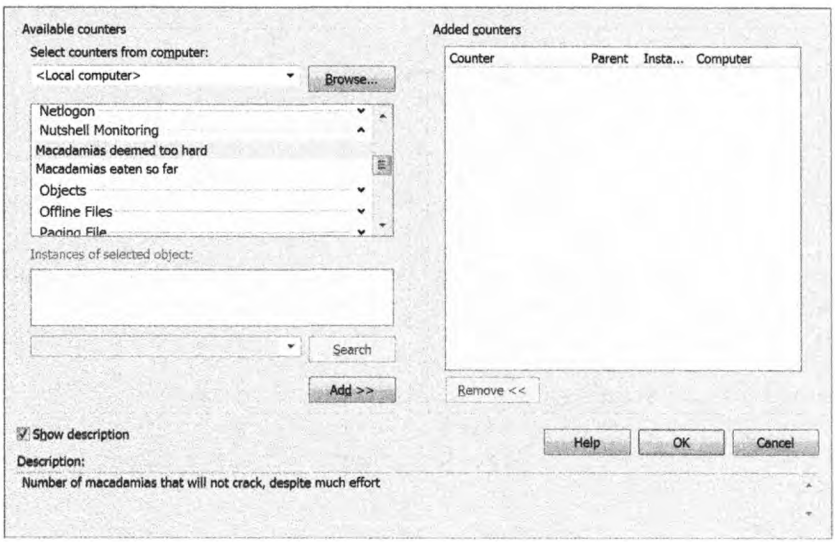



***Рис. 13.1.*** *Специальные счетчики производительности*

Если позже понадобится определить дополнительные счетчики в той же самой категории, то старая категория должна быть сначала удалена вызовом метода `PerformanceCounterCategory.Delete`.

Создание и удаление счетчиков производительности требует наличия административных полномочий. По этой причине такие действия выполняются как часть процесса установки приложения.

После того, как счетчик создан, его значение можно обновить, создав экземпляр `PerformanceCounter`, установив его свойство `ReadOnly` в `false` и затем установив его свойство `RawValue`. Для обновления существующего значения можно также применять методы `Increment` и `IncrementBy`:

```
string category = "Nutshell Monitoring";
string eatenPerMin = "Macadamias eaten so far";

using (PerformanceCounter pc = new PerformanceCounter (category,
                                                       eatenPerMin, ""))
{
```

```
    pc.ReadOnly = false;
    pc.RawValue = 1000;
    pc.Increment();
    pc.IncrementBy (10);
    Console.WriteLine (pc.NextValue());  // 1011
  }
```

# Класс Stopwatch

Класс Stopwatch предлагает удобный механизм для измерения времени выполнения. Класс Stopwatch использует механизм с самым высоким разрешением, какое только обеспечивается операционной системой и оборудованием; обычно разрешение составляет меньше одной микросекунды. (По контрасту с ним свойства DateTime.Now и Environment.TickCount поддерживают разрешение около 15 миллисекунд.)

Для работы с классом Stopwatch необходимо вызвать метод StartNew – в результате создается новый экземпляр Stopwatch и запускается измерение времени. (В качестве альтернативы экземпляр Stopwatch можно создать вручную и затем вызвать метод Start.) Свойство Elapsed возвращает интервал пройденного времени в виде структуры TimeSpan:

```
Stopwatch s = Stopwatch.StartNew();
System.IO.File.WriteAllText ("test.txt", new string ('*', 30000000));
Console.WriteLine (s.Elapsed);  // 00:00:01.4322661
```

Класс Stopwatch также открывает доступ к свойству ElapsedTicks, которое возвращает количество пройденных "тиков" как значение long. Чтобы преобразовать тики в секунды, разделите полученное значение на StopWatch.Frequency. Есть также свойство ElapsedMilliseconds, которое часто оказывается наиболее удобным.

Вызов метода Stop фиксирует значения свойств Elapsed и ElapsedTicks. Никакого фонового действия, связанного с "выполнением" Stopwatch, не предусмотрено, а потому вызов метода Stop является необязательным.

# 14

# Параллелизм и асинхронность

Большинству приложений приходится иметь дело сразу с несколькими активностями, происходящими одновременно (*параллелизм*). Настоящую главу мы начнем с рассмотрения важнейших предпосылок, а именно – основ многопоточности и задач, после чего подробно обсудим принципы асинхронности и асинхронные функции C#.

В главе 22 мы продолжим более детальный анализ многопоточности, а в главе 23 раскроем связанную тему параллельного программирования.

## Введение

Ниже приведены самые распространенные сценарии применения параллелизма.

- *Написание отзывчивых пользовательских интерфейсов.* Для обеспечения приемлемого времени отклика в приложениях WPF, мобильных приложениях и приложениях Windows Forms длительно выполняющиеся задачи должны запускаться параллельно с кодом, реализующим пользовательский интерфейс.
- *Обеспечение одновременной обработки запросов.* Клиентские запросы могут поступать на сервер одновременно, а потому они должны обрабатываться параллельно для обеспечения масштабируемости. В случае использования инфраструктуры ASP.NET, WCF или Web Services платформа .NET Framework делает это автоматически. Тем не менее, вы по-прежнему должны заботиться о разделяемом состоянии (например, учитывать последствия применения статических переменных для кеширования).
- *Параллельное программирование.* Код, в котором присутствуют интенсивные вычисления, может выполняться быстрее на многоядерных/многопроцессорных компьютерах, если рабочая нагрузка распределяется между ядрами (данной теме посвящена глава 23).
- *Упреждающее выполнение.* На многоядерных машинах иногда удается улучшить производительность, предсказывая то, что возможно понадобится сделать, и выполняя это действие заранее. В LINQPad такой прием используется для ускорения создания новых запросов. Вариацией может быть запуск нескольких алгоритмов параллельно для решения одной и той же задачи. Тот из них, который завершится первым, "выиграет" – прием эффективен, когда нельзя узнать заранее, какой алгоритм будет выполняться быстрее всех.

Общий механизм, с помощью которого программа может выполнять код одновременно, называется *многопоточностью*. Многопоточность поддерживается как средой CLR, так и операционной системой (ОС), и в рамках параллелизма является фундаментальной концепцией. Таким образом, жизненно важно четко понимать основы многопоточной обработки и в особенности влияние потоков на *разделяемое состояние*.

# Многопоточная обработка

*Поток* – это путь выполнения, который может проходить независимо от других таких путей.

Каждый поток запускается внутри процесса ОС, который предоставляет изолированную среду для выполнения программы. В *однопоточной* программе внутри изолированной среды процесса функционирует только один поток, поэтому он получает монопольный доступ к среде. В *многопоточной* программе внутри единственного процесса запускается множество потоков, разделяя одну и ту же среду выполнения (скажем, память). Отчасти это одна из причин, почему полезна многопоточность: например, один поток может извлекать данные в фоновом режиме, в то время как другой поток – отображать их по мере поступления. Такие данные называются *разделяемым состоянием*.

## Создание потока

В приложениях UWP нельзя создавать и запускать потоки напрямую; взамен это должно делаться через задачи (см. раздел "Задачи" далее в главе). Задачи добавляют уровень косвенности, который усложняет изучение, так что лучше всего начинать с консольных приложений (или LINQPad) и создавать потоки напрямую до тех пор, пока вы не освоитесь с тем, как они работают.

*Клиентская* программа (консольная, WPF, UWP или Windows Forms) запускается в единственном потоке, который создается автоматически операционной системой ("главный" поток). Здесь он и будет существовать как однопоточное приложение, если только вы не создадите дополнительные потоки (прямо или косвенно)[1].

Создать и запустить новый поток можно за счет создания объекта `Thread` и вызова его метода `Start`. Простейший конструктор `Thread` принимает делегат `ThreadStart`: метод без параметров, который указывает, где должно начинаться выполнение. Например:

```
// Напоминание: во всех примерах главы предполагается
// импортирование следующих пространств имен:
using System;
using System.Threading;
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (WriteY);  // Начать новый поток,
    t.Start();                       // выполняющий WriteY.
    // Одновременно делать что-то в главном потоке.
    for (int i = 0; i < 1000; i++) Console.Write ("x");
  }
```

[1] "За кулисами" CLR создает другие потоки, предназначенные для сборки мусора и финализации.

```
  static void WriteY()
  {
    for (int i = 0; i < 1000; i++) Console.Write ("y");
  }
}
// Типичный вывод:
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

Главный поток создает новый поток `t`, в котором запускает метод, многократно выводящий символ *y*. В то же самое время главный поток многократно выводит символ *x* (рис. 14.1). На компьютере с одноядерным процессором ОС должна выделять каждому потоку кванты времени (обычно размером 20 миллисекунд в среде Windows) для эмуляции параллелизма, что дает в результате повторяющиеся блоки вывода *x* и *y*. На многоядерной или многопроцессорной машине два потока могут выполняться по-настоящему параллельно (конкурируя с другими активными процессами в системе), хотя в рассматриваемом примере все равно будут получаться повторяющиеся блоки вывода *x* и *y* из-за тонкостей работы механизма, которым класс `Console` обрабатывает параллельные запросы.

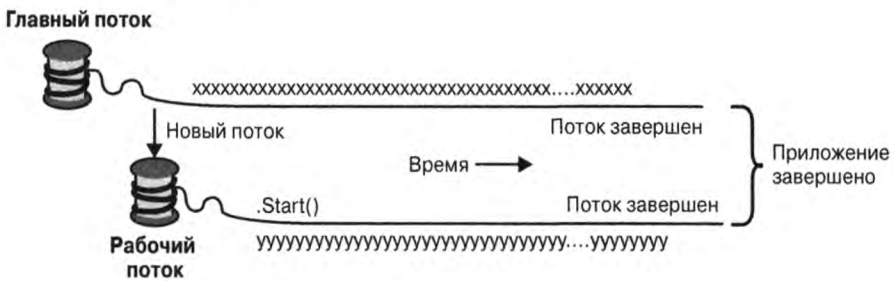



***Рис. 14.1.** Начало нового потока*

Говорят, что поток *вытесняется* в точках, где его выполнение пересекается с выполнением кода в другом потоке. К этому термину часто прибегают при объяснении, почему что-то пошло не так, как было задумано!

После запуска свойство `IsAlive` потока возвращает `true` до тех пор, пока не будет достигнута точка, где поток завершается. Поток заканчивается, когда завершает выполнение делегат, переданный конструктору класса `Thread`. После завершения поток не может быть запущен повторно.

Каждый поток имеет свойство `Name`, которое можно установить для содействия отладке. Это особенно полезно в Visual Studio, т.к. имя потока отображается в окне Threads (Потоки) и в панели инструментов Debug Location (Местоположение отладки). Установить имя потока можно только один раз; попытки изменить его позже приведут к генерации исключения.

Статическое свойство `Thread.CurrentThread` возвращает поток, выполняющийся в текущее время:

```
Console.WriteLine (Thread.CurrentThread.Name);
```

## Join и Sleep

С помощью вызова метода Join можно организовать ожидание окончания другого потока:

```
static void Main()
{
  Thread t = new Thread (Go);
  t.Start();
  t.Join();
  Console.WriteLine ("Thread t has ended!");  // Поток t завершен!
}
static void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }
```

Код выводит на консоль символ y тысячу раз и затем сразу же строку Thread t has ended!. При вызове метода Join можно указывать тайм-аут, выраженный в миллисекундах или в виде структуры TimeSpan. Тогда метод будет возвращать true, если поток был завершен, или false, если истекло время тайм-аута.

Метод Thread.Sleep приостанавливает текущий поток на заданный период:

```
Thread.Sleep (TimeSpan.FromHours (1));      // Ожидать 1 час
Thread.Sleep (500);                         // Ожидать 500 миллисекунд
```

Вызов Thread.Sleep(0) немедленно прекращает текущий квант времени потока, добровольно передавая контроль над центральным процессором (ЦП) другим потокам. Метод Thread.Yield() делает то же самое, но уступает контроль только потокам, функционирующим на *том же самом* процессоре.

Вызов Sleep(0) или Yield() в производственном коде иногда полезен для расширенной настройки производительности. Это также великолепный диагностический инструмент для поиска проблем, связанных с безопасностью к потокам: если вставка вызова Thread.Yield() в любое место кода нарушает работу программы, то в ней почти наверняка присутствует ошибка.

На время ожидания Sleep или Join поток *блокируется*.

## Блокировка

Поток считается *заблокированным*, если его выполнение приостановлено по некоторой причине, такой как вызов метода Sleep или ожидание завершения другого потока через вызов Join. Заблокированный поток немедленно *уступает* свой квант процессорного времени и далее не потребляет процессорное время, пока удовлетворяется условие блокировки. Проверить, заблокирован ли поток, можно с помощью его свойства ThreadState:

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```

Свойство ThreadState является перечислением флагов, комбинирующим три "уровня" данных в побитовой манере. Однако большинство значений являются избыточными, неиспользуемыми или устаревшими. Следующий расширяющий метод ограничивает ThreadState одним из четырех полезных значений: Unstarted, Running, WaitSleepJoin и Stopped:

```
public static ThreadState Simplify (this ThreadState ts)
{
  return ts & (ThreadState.Unstarted |
               ThreadState.WaitSleepJoin |
               ThreadState.Stopped);
}
```

Свойство `ThreadState` удобно для диагностических целей, но непригодно для синхронизации, т.к. состояние потока может изменяться в промежутке между проверкой `ThreadState` и обработкой данной информации.

Когда поток блокируется или деблокируется, ОС производит *переключение контекста*. С ним связаны небольшие накладные расходы, обычно составляющие одну или две микросекунды.

### Интенсивный ввод-вывод или интенсивные вычисления

Операция, которая бо́льшую часть своего времени тратит на ожидание, пока что-то произойдет, называется операцией с *интенсивным вводом-выводом*; примером может служить загрузка веб-страницы или вызов метода `Console.ReadLine`. (Операции с интенсивным вводом-выводом обычно включают в себя ввод или вывод, но это не жесткое требование: вызов метода `Thread.Sleep` также считается операцией с интенсивным вводом-выводом.) И напротив, операция, которая бо́льшую часть своего времени затрачивает на выполнение вычислений с привлечением ЦП, называется операцией с *интенсивными вычислениями*.

### Блокирование или зацикливание

Операция с интенсивным вводом-выводом работает одним из двух способов. Она либо *синхронно* ожидает завершения определенной операции в текущем потоке (такой как `Console.ReadLine`, `Thread.Sleep` или `Thread.Join`), либо работает *асинхронно*, инициируя обратный вызов, когда интересующая операция завершается спустя какое-то время (более подробно об этом позже).

Операции с интенсивным вводом-выводом, которые ожидают синхронным образом, бо́льшую часть своего времени тратят на блокирование потока. Они также могут периодически "прокручиваться" в цикле:

```
while (DateTime.Now < nextStartTime)
  Thread.Sleep (100);
```

Оставляя в стороне тот факт, что существуют более удачные средства (вроде таймеров и сигнализирующих конструкций), еще одна возможность предусматривает зацикливание потока:

```
while (DateTime.Now < nextStartTime);
```

В общем случае это очень неэкономное расходование процессорного времени: среда CLR и ОС предполагают, что поток выполняет важные вычисления, и соответствующим образом выделяют ресурсы. В сущности, мы превращаем код, который должен быть операцией с интенсивным вводом-выводом, в операцию с интенсивными вычислениями.

Относительно вопроса зацикливания или блокирования следует отметить пару нюансов. Во-первых, *очень кратковременное* зацикливание может быть эффективным, когда ожидается скорое (возможно в пределах нескольких микросекунд) удовлетворение некоторого условия, поскольку оно избегает

накладных расходов и задержки, связанной с переключением контекста. Инфраструктура .NET Framework предлагает специальные методы и классы для содействия зацикливанию (см. информацию по ссылке SpinLock and SpinWait на странице http://albahari.com/threading/).

Во-вторых, затраты на блокирование не являются *нулевыми*. Дело в том, что за время своего существования каждый поток связывает около 1 Мбайт памяти и служит источником текущих накладных расходов на администрирование со стороны среды CLR и ОС. По этой причине блокирование может быть ненадежным в контексте программ с интенсивным вводом-выводом, которые нуждаются в поддержке сотен или тысяч параллельных операций. Взамен такие программы должны использовать подход, основанный на обратных вызовах, что полностью освободит поток на время ожидания. Таково (отчасти) целевое назначение асинхронных шаблонов, которые мы обсудим позже.

## Локальное или разделяемое состояние

Среда CLR назначает каждому потоку собственный стек в памяти, так что локальные переменные хранятся отдельно. В следующем примере мы определяем метод с локальной переменной, после чего вызываем его одновременно в главном потоке и во вновь созданном потоке:

```
static void Main()
{
  new Thread (Go).Start();     // Вызвать Go в новом потоке
  Go();                        // Вызвать Go в главном потоке
}
static void Go()
{
  // Объявить и использовать локальную переменную cycles
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

В стеке каждого потока создается отдельная копия переменной cycles, так что вывод вполне предсказуемо содержит десять знаков вопроса. Потоки разделяют данные, если они имеют общую ссылку на один и тот же экземпляр:

```
class ThreadTest
{
  bool _done;
  static void Main()
  {
    ThreadTest tt = new ThreadTest();  // Создать общий экземпляр
    new Thread (tt.Go).Start();
    tt.Go();
  }
  void Go()        // Обратите внимание, что это метод экземпляра
  {
    if (!_done) { _done = true; Console.WriteLine ("Done"); }
  }
}
```

Поскольку оба потока вызывают метод Go на одном и том же экземпляре ThreadTest, они разделяют поле _done. В результате слово Done выводится один раз, а не два.

Локальные переменные, захваченные лямбда-выражением или анонимным делегатом, преобразуются компилятором в поля, поэтому они также могут быть разделяемыми:

```
class ThreadTest
{
  static void Main()
  {
    bool done = false;
    ThreadStart action = () =>
    {
      if (!done) { done = true; Console.WriteLine ("Done"); }
    };
    new Thread (action).Start();
    action();
  }
}
```

Статические поля предлагают еще один способ разделения данных между потоками:

```
class ThreadTest
{
  static bool _done;      // Статические поля разделяются между всеми
                          // потоками в том же самом домене приложения.
  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }
  static void Go()
  {
    if (!_done) { _done = true; Console.WriteLine ("Done"); }
  }
}
```

Все три примера иллюстрируют еще одну ключевую концепцию: безопасность в отношении потоков (или наоборот – ее отсутствие). Вывод в действительности не определен: возможно (хотя и маловероятно), что слово Done будет выведено дважды. Однако если мы поменяем местами порядок следования операторов в методе Go, то вероятность двукратного вывода слова Done значительно возрастет:

```
static void Go()
{
  if (!_done) { Console.WriteLine ("Done"); _done = true; }
}
```

Проблема связана с тем, что пока в одном потоке оценивается оператор if, во втором потоке выполняется вызов WriteLine – до того, как он получит шанс установить поле _done в true.

Приведенный пример демонстрирует одну из многочисленных ситуаций, в которых *разделяемое записываемое состояние* может привести к возникновению определенной разновидности несистематических ошибок, характерных для многопоточности. В следующем разделе мы покажем, как с помощью блокировки исправить программу; тем не менее, лучше по возможности вообще избегать применения разделяемого состояния. Позже мы объясним, как в этом могут помочь шаблоны асинхронного программирования.

## Блокировка и безопасность потоков

Блокировка и безопасность в отношении потоков являются обширными темами. Полное их обсуждение приведено в разделах "Монопольное блокирование" и "Блокирование и безопасность к потокам" главы 22.

Исправить предыдущий пример можно, получив *монопольную блокировку* на период чтения и записи разделяемого поля. Для этой цели в языке C# предусмотрен оператор `lock`:

```
class ThreadSafe
{
  static bool _done;
  static readonly object _locker = new object();
  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }
  static void Go()
  {
    lock (_locker)
    {
      if (!_done) { Console.WriteLine ("Done"); _done = true; }
    }
  }
}
```

Когда два потока одновременно соперничают за блокировку (что может возникать с любым объектом ссылочного типа; в рассматриваемом случае `_locker`), один из потоков ожидает, или блокируется, до тех пор, пока блокировка не станет доступной. В таком случае гарантируется, что только один поток может войти в данный блок кода за раз, и строка `Done` будет выведена лишь однократно. Код, защищенный подобным образом — от неопределенности в многопоточном контексте — называется *безопасным в отношении потоков*.

Даже действие автоинкрементирования переменной не является безопасным к потокам: выражение `x++` выполняется на лежащем в основе процессоре как отдельные операции чтения, инкремента и записи. Таким образом, если два потока выполняют `x++` одновременно за пределами блокировки, то переменная `x` в итоге может быть инкрементирована один раз, а не два (или, что еще хуже, в определенных обстоятельствах переменная `x` может быть вообще *разрушена*, получив смесь битов старого и нового содержимого).

Блокировка не является панацеей для обеспечения безопасности потоков — довольно легко забыть заблокировать доступ к полю, и тогда блокировка сама может создать проблемы (наподобие состояния взаимоблокировки).

Хорошим примером применения блокировки может служить доступ к разделяемому кешу внутри памяти для часто используемых объектов базы данных в приложении ASP.NET. Приложение такого вида очень просто заставить работать правильно без возникновения взаимоблокировки. Пример будет приведен в разделе "Безопасность к потокам в серверах приложений" главы 22.

## Передача данных потоку

Иногда требуется передать аргументы начальному методу потока. Проще всего это сделать с использованием лямбда-выражения, которое вызывает данный метод с желаемыми аргументами:

```
static void Main()
{
  Thread t = new Thread ( () => Print ("Hello from t!") );
  t.Start();
}
static void Print (string message) { Console.WriteLine (message); }
```

Такой подход позволяет передавать методу любое количество аргументов. Можно даже поместить всю реализацию в лямбда-функцию с множеством операторов:

```
new Thread (() =>
{
  Console.WriteLine ("I'm running on another thread!");
  Console.WriteLine ("This is so easy!");
}).Start();
```

В версиях, предшествующих C# 3.0, лямбда-выражения не существовали. Таким образом, вы также могли сталкиваться со старым подходом, предусматривающим передачу аргумента методу `Start` класса `Thread`:

```
static void Main()
{
  Thread t = new Thread (Print);
  t.Start ("Hello from t!");
}
static void Print (object messageObj)
{
  string message = (string) messageObj;  // Здесь необходимо приведение
  Console.WriteLine (message);
}
```

Код работает потому, что конструктор класса `Thread` перегружен для приема одного из двух делегатов:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart (object obj);
```

Ограничение делегата `ParameterizedThreadStart` в том, что он принимает только один аргумент. И поскольку аргумент имеет тип `object`, обычно требуется приведение.

### Лямбда-выражения и захваченные переменные

Как уже должно быть понятно, лямбда-выражение является наиболее удобным и мощным способом передачи данных потоку. Однако следует соблюдать осторожность, чтобы случайно не изменить *захваченные переменные* после запуска потока. Например, рассмотрим следующий код:

```
for (int i = 0; i < 10; i++)
  new Thread (() => Console.Write (i)).Start();
```

Вывод будет недетерминированным! Вот типичный результат:

```
0223557799
```

Проблема в том, что на протяжении всего времени жизни цикла переменная `i` ссылается на *ту же самую* ячейку в памяти. Следовательно, каждый поток вызывает метод `Console.Write` с переменной, значение которой может измениться по мере его выполнения! Решение заключается в применении временной переменной, как показано ниже:

```
for (int i = 0; i < 10; i++)
{
  int temp = i;
  new Thread (() => Console.Write (temp)).Start();
}
```

Теперь все цифры от 0 до 9 будут выводиться в точности по одному разу. (*Порядок* вывода по-прежнему не определен, т.к. потоки могут запускаться в неопределимые моменты времени.)

Данная проблема аналогична проблеме, описанной в разделе "Захваченные переменные" главы 8. Она в основном обусловлена правилами языка C# для захвата переменных внутри циклов `for` в многопоточном сценарии.

Указанная проблема также характерна для циклов `foreach` в версиях, предшествующих C# 5.

Переменная `temp` является локальной по отношению к каждой итерации цикла. Таким образом, каждый поток захватывает отличающуюся ячейку памяти, и проблемы не возникают. Проблему в приведенном ранее коде проще проиллюстрировать с помощью показанного далее примера:

```
string text = "t1";
Thread t1 = new Thread ( () => Console.WriteLine (text) );
text = "t2";
Thread t2 = new Thread ( () => Console.WriteLine (text) );
t1.Start();
t2.Start();
```

Поскольку оба лямбда-выражения захватывают одну и ту же переменную `text`, строка `t2` выводится дважды.

## Обработка исключений

Любые блоки `try/catch/finally`, действующие во время создания потока, не играют никакой роли в потоке, когда он начинает свое выполнение. Взгляните на следующую программу:

```
public static void Main()
{
  try
  {
    new Thread (Go).Start();
  }
  catch (Exception ex)
  {
    // Сюда мы никогда не попадем!
    Console.WriteLine ("Exception!");  // Исключение!
  }
}
static void Go() { throw null; }  // Генерирует исключение NullReferenceException
```

Оператор try/catch здесь безрезультатен, и вновь созданный поток будет обременен необработанным исключением NullReferenceException. Такое поведение имеет смысл, если принять во внимание тот факт, что каждый поток обладает независимым путем выполнения.

Чтобы исправить ситуацию, обработчик событий потребуется переместить внутрь метода Go:

```
public static void Main()
{
  new Thread (Go).Start();
}
static void Go()
{
  try
  {
    ...
    throw null;  // Исключение NullReferenceException будет перехвачено ниже
    ...
  }
  catch (Exception ex)
  {
    // Обычно необходимо зарегистрировать исключение в журнале
    // и/или сигнализировать другому потоку об отсоединении
    ...
  }
}
```

В производственных приложениях необходимо предусмотреть обработчики исключений для всех методов входа в потоки — в точности как это делается в главном потоке (обычно на более высоком уровне в стеке выполнения). Необработанное исключение приведет к прекращению работы всего приложения, да еще и с отображением безобразного диалогового окна!

При написании таких блоков обработки исключений вы редко будете *игнорировать* ошибку: обычно вы зарегистрируете в журнале подробности исключения и возможно отобразите диалоговое окно, позволяющее пользователю автоматически отправить подробные сведения веб-серверу. Затем вероятно вы решите перезапустить приложение, поскольку существует возможность того, что непредвиденное исключение оставило приложение в недопустимом состоянии.

## Централизованная обработка исключений

В приложениях WPF, UWP и Windows Forms можно подписываться соответственно на "глобальные" события обработки исключений, Application.DispatcherUnhandledException и Application.ThreadException. Они инициируются после возникновения необработанного исключения в любой части программы, которая вызвана в цикле сообщений (сказанное относится ко всему коду, выполняющемуся в главном потоке, пока активен экземпляр Application). Прием полезен в качестве резервного средства для регистрации и сообщения об ошибках (хотя он неприменим для необработанных исключений, которые возникают в созданных вами потоках, не относящихся к пользовательскому интерфейсу). Обработка упомянутых событий предотвращает аварийное завершение программы, хотя впоследствии может быть принято решение о ее перезапуске во избежание потенциального разрушения состояния, к которому может привести необработанное исключение.

Событие `AppDomain.CurrentDomain.UnhandledException` инициируется любым необработанным исключением, возникающим в любом потоке, но, начиная с версии 2.0, среда CLR принудительно прекращает работу приложения после завершения вашего обработчика исключений. Тем не менее, прекращение работы можно предотвратить, добавив в файл конфигурации приложения следующий код:

```
<configuration>
  <runtime>
    <legacyUnhandledExceptionPolicy enabled="1" />
  </runtime>
</configuration>
```

Это может оказаться полезным в программах, содержащих множество доменов приложений (глава 24). Если необработанное приложение возникло в домене приложения, который не является стандартным, то такой домен можно уничтожить и создать заново вместо того, чтобы перезапускать целое приложение.

## Потоки переднего плана или фоновые потоки

По умолчанию потоки, создаваемые явно, являются *потоками переднего плана.* Потоки переднего плана удерживают приложение в активном состоянии до тех пор, пока хотя бы один из них выполняется, но *фоновые потоки* этого не делают. Как только все потоки переднего плана завершают свою работу, заканчивается и приложение, а любые все еще выполняющиеся фоновые потоки будут принудительно прекращены.

Состояние переднего плана или фоновое состояние потока не имеет никакого отношения к его *приоритету* (выделению времени на выполнение).

Выяснить либо изменить фоновое состояние потока можно с использованием его свойства `IsBackground`:

```
static void Main (string[] args)
{
  Thread worker = new Thread ( () => Console.ReadLine() );
  if (args.Length > 0) worker.IsBackground = true;
  worker.Start();
}
```

Если запустить такую программу без аргументов, тогда рабочий поток предполагает, что она находится в фоновом состоянии, и будет ожидать в операторе `ReadLine` нажатия пользователем клавиши <Enter>. Тем временем главный поток завершается, но приложение остается запущенным, потому что поток переднего плана все еще активен. С другой стороны, если методу `Main` передается аргумент, то рабочему потоку назначается фоновое состояние, и программа завершается почти сразу после завершения главного потока (прекращая выполнение метода `ReadLine`).

Когда процесс прекращает работу подобным образом, любые блоки `finally` в стеке выполнения фоновых потоков пропускаются. Если программа задействует блоки `finally` (или `using`) для проведения очистки вроде удаления временных файлов, то вы можете избежать этого, явно ожидая такие фоновые потоки вплоть до завершения приложения, либо за счет присоединения потока, либо с помощью сигнализирующей конструкции (см. раздел "Передача сигналов" далее в главе). В любом случае должен быть указан тайм-аут, чтобы можно было уничтожить поток, который отказывается завершаться, иначе приложение не сможет быть нормально закрыто без привлечения пользователем диспетчера задач.

Потоки переднего плана не требуют такой обработки, но вы должны позаботиться о том, чтобы избежать ошибок, которые могут привести к отказу завершения потока. Обычной причиной отказа в корректном завершении приложений является наличие активных фоновых потоков.

## Приоритет потока

Свойство `Priority` потока определяет, сколько времени на выполнение получит данный поток относительно других активных потоков в ОС, со следующей шкалой значений:

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

Это становится важным, когда одновременно активно несколько потоков. Увеличение приоритета потока должно производиться осторожно, т.к. может привести к торможению других потоков. Если нужно, чтобы поток имел больший приоритет, чем потоки в *других* процессах, тогда потребуется также увеличить приоритет процесса с применением класса `Process` из пространства имен `System.Diagnostics`:

```
using (Process p = Process.GetCurrentProcess())
  p.PriorityClass = ProcessPriorityClass.High;
```

Прием может нормально работать для потоков, не относящихся к пользовательскому интерфейсу, которые выполняют минимальную работу и нуждаются в низкой задержке (в возможности реагировать очень быстро). В приложениях с обильными вычислениями (особенно в тех, которые имеют пользовательский интерфейс) увеличение приоритета процесса может приводить к торможению других процессов и замедлению работы всего компьютера.

## Передача сигналов

Иногда нужно, чтобы поток ожидал получения уведомления (уведомлений) от другого потока (потоков). Это называется *передачей сигналов*. Простейшей сигнализирующей конструкцией является класс `ManualResetEvent`. Вызов метода `WaitOne` класса `ManualResetEvent` блокирует текущий поток до тех пор, пока другой поток не "откроет" сигнал, вызвав метод `Set`. В приведенном ниже примере мы запускаем поток, который ожидает события `ManualResetEvent`. Он остается заблокированным в течение двух секунд до тех пор, пока главный поток не выдаст *сигнал*:

```
var signal = new ManualResetEvent (false);
new Thread (() =>
{
  Console.WriteLine ("Waiting for signal...");   // Ожидание сигнала...
  signal.WaitOne();
  signal.Dispose();
  Console.WriteLine ("Got signal!");             // Сигнал получен!
}).Start();
Thread.Sleep(2000);
signal.Set();                                    // "Открыть" сигнал
```

После вызова метода `Set` сигнал остается открытым; для его закрытия понадобится вызвать метод `Reset`.

Класс `ManualResetEvent` — одна из нескольких сигнализирующих конструкций, предоставляемых средой CLR; все они подробно рассматриваются в главе 22.

## Многопоточность в обогащенных клиентских приложениях

В приложениях WPF, UWP и Windows Forms выполнение длительных по времени операций в главном потоке снижает отзывчивость приложения, потому что главный поток обрабатывает также цикл сообщений, который отвечает за визуализацию и поддержку событий клавиатуры и мыши.

Популярный подход предусматривает настройку "рабочих" потоков для выполнения длительных по времени операций. Код в рабочем потоке запускает длительную операцию и по ее завершении обновляет пользовательский интерфейс. Тем не менее, все обогащенные клиентские приложения поддерживают потоковую модель, в которой элементы управления пользовательского интерфейса могут быть доступны только из создавшего их потока (обычно главного потока пользовательского интерфейса). Нарушение данного правила приводит либо к непредсказуемому поведению, либо к генерации исключения.

Следовательно, когда нужно обновить пользовательский интерфейс из рабочего потока, запрос должен быть перенаправлен потоку пользовательского интерфейса (формально это называется *маршализацией*). Вот как выглядит низкоуровневый способ реализации такого действия (позже мы обсудим другие решения, которые на нем основаны):

- в приложении WPF вызовите метод `BeginInvoke` или `Invoke` на объекте `Dispatcher` элемента;
- в приложении UWP вызовите метод `RunAsync` или `Invoke` на объекте `Dispatcher`;
- в приложении Windows Forms вызовите метод `BeginInvoke` или `Invoke` на элементе управления.

Все упомянутые методы принимают делегат, ссылающийся на метод, который требуется запустить. Методы `BeginInvoke`/`RunAsync` работают путем постановки этого делегата в *очередь сообщений* потока пользовательского интерфейса (та же очередь, которая обрабатывает события, поступающие от клавиатуры, мыши и таймера). Метод `Invoke` делает то же самое, но затем блокируется до тех пор, пока сообщение не будет прочитано и обработано потоком пользовательского интерфейса. По указанной причине метод `Invoke` позволяет получить возвращаемое значение из метода. Если возвращаемое значение не требуется, то методы `BeginInvoke`/`RunAsync` предпочтительнее из-за того, что они не блокируют вызывающий компонент и не привносят возможность возникновения взаимоблокировки (см. раздел "Взаимоблокировки" в главе 22).

Вы можете представлять себе, что при вызове метода `Application.Run` выполняется следующий псевдокод:

```
while (приложение не завершено)
{
  Ожидать появления чего-нибудь в очереди сообщений
  Что-то получено: к какому виду сообщений оно относится?
    Сообщение клавиатуры/мыши -> запустить обработчик событий
    Пользовательское сообщение BeginInvoke -> выполнить делегат
    Пользовательское сообщение Invoke ->
                          выполнить делегат и отправить результат
}
```

Цикл такого вида позволяет рабочему потоку маршализировать делегат для выполнения в потоке пользовательского интерфейса.

В целях демонстрации предположим, что имеется окно WPF с текстовым полем по имени txtMessage, содержимое которого должно быть обновлено рабочим потоком после выполнения длительной задачи (эмулируемой с помощью вызова метода Thread.Sleep). Ниже приведен необходимый код:

```
partial class MyWindow : Window
{
  public MyWindow()
  {
    InitializeComponent();
    new Thread (Work).Start();
  }
  void Work()
  {
    Thread.Sleep (5000);  // Эмулировать длительно выполняющуюся задачу
    UpdateMessage ("The answer");
  }
  void UpdateMessage (string message)
  {
    Action action = () => txtMessage.Text = message;
    Dispatcher.BeginInvoke (action);
  }
}
```

После запуска показанного кода немедленно появляется окно. Спустя пять секунд текстовое поле обновляется. Для случая Windows Forms код будет похож, но только в нем вызывается метод BeginInvoke объекта Form:

```
void UpdateMessage (string message)
{
  Action action = () => txtMessage.Text = message;
  this.BeginInvoke (action);
}
```

### Множество потоков пользовательского интерфейса

Допускается иметь множество потоков пользовательского интерфейса, если каждый из них владеет своим окном. Основным сценарием может служить приложение с несколькими высокоуровневыми окнами, которое часто называют приложением с *однодокументным интерфейсом* (Single Document Interface – SDI), например, Microsoft Word. Каждое окно SDI обычно отображает себя как отдельное "приложение" в панели задач и по большей части оно функционально изолировано от других окон SDI. За счет предоставления каждому такому окну собственного потока пользовательского интерфейса окна становятся более отзывчивыми.

## Контексты синхронизации

В пространстве имен System.ComponentModel определен абстрактный класс SynchronizationContext, который делает возможным обобщение маршализации потоков.

В обогащенных API-интерфейсах для мобильных и настольных приложений (UWP, WPF и Windows Forms) определены и созданы экземпляры подклассов

SynchronizationContext, которые можно получить через статическое свойство SynchronizationContext.Current (при выполнении в потоке пользовательского интерфейса). Захват этого свойства позволяет позже "отправлять" сообщения элементам управления пользовательского интерфейса из рабочего потока:

```
partial class MyWindow : Window
{
  SynchronizationContext _uiSyncContext;
  public MyWindow()
  {
    InitializeComponent();
    // Захватить контекст синхронизации для текущего потока
    // пользовательского интерфейса:
    _uiSyncContext = SynchronizationContext.Current;
    new Thread (Work).Start();
  }
  void Work()
  {
    Thread.Sleep (5000);  // Эмулировать длительно выполняющуюся задачу
    UpdateMessage ("The answer");
  }
  void UpdateMessage (string message)
  {
    // Маршализировать делегат потоку пользовательского интерфейса:
    _uiSyncContext.Post (_ => txtMessage.Text = message, null);
  }
}
```

Удобство в том, что один и тот же подход работает со всеми обогащенными API-интерфейсами (класс SynchronizationContext также имеет специализацию для ASP.NET, где он играет более тонкую роль, обеспечивая последовательную обработку страничных событий в соответствии с асинхронными операциями и предохраняя HttpContext).

Вызов метода Post эквивалентен вызову BeginInvoke на объекте Dispatcher или Control; есть также метод Send, который является эквивалентом Invoke.

В версии .NET Framework 2.0 был введен класс BackgroundWorker, который использует класс SynchronizationContext для упрощения работы по управлению рабочими потоками в обогащенных клиентских приложениях. Позже класс BackgroundWorker стал избыточным из-за появления классов задач и асинхронных функций, которые, как вы увидите, также имеют дело с SynchronizationContext.

## Пул потоков

Всякий раз, когда запускается поток, несколько сотен микросекунд тратится на организацию таких элементов, как новый стек локальных переменных. Снизить эти накладные расходы позволяет *пул потоков*, предлагая накопитель заранее созданных многократно применяемых потоков. Организация пула потоков жизненно важна для эффективного параллельного программирования и реализации мелкомодульного параллелизма; пул потоков позволяет запускать короткие операции без накладных расходов, связанных с начальной настройкой потока.

При использовании потоков из пула следует учитывать несколько моментов.

- Невозможность установки свойства `Name` потока из пула затрудняет отладку (хотя при отладке в окне Threads среды Visual Studio к потоку можно присоединять описание).
- Потоки из пула всегда являются фоновыми.
- Блокирование потоков из пула может привести к снижению производительности (см. раздел "Чистота пула потоков" далее в главе).

Приоритет потока из пула можно свободно изменять — когда поток возвратится обратно в пул, будет восстановлен его первоначальный приоритет.

Для выяснения, является ли текущий поток потоком из пула, предназначено свойство `Thread.CurrentThread.IsThreadPoolThread`.

### Вход в пул потоков

Простейший способ явного запуска какого-то кода в потоке из пула предполагает применение метода `Task.Run` (мы рассмотрим прием более подробно в следующем разделе):

```
// Класс Task находится в пространство имен System.Threading.Tasks
Task.Run (() => Console.WriteLine ("Hello from the thread pool"));
```

Поскольку до выхода версии .NET Framework 4.0 классы задач не существовали, общепринятой альтернативой являлся вызов метода `ThreadPool.QueueUserWorkItem`:

```
ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
```

Перечисленные ниже компоненты неявно используют пул потоков:

- серверы приложений WCF, Remoting, ASP.NET и ASMX Web Services;
- классы `System.Timers.Timer` и `System.Threading.Timer`;
- конструкции параллельного программирования, которые будут описаны в главе 23;
- класс `BackgroundWorker` (теперь избыточный);
- асинхронные делегаты (также теперь избыточные).

### Чистота пула потоков

Пул потоков содействует еще одной функции, которая гарантирует то, что временный излишек интенсивной вычислительной работы не приведет к *превышению лимита* ЦП. Превышение лимита — это условие, при котором активных потоков имеется больше, чем ядер ЦП, и операционная система вынуждена выделять потокам кванты времени. Превышение лимита наносит ущерб производительности, т.к. выделение квантов времени требует интенсивных переключений контекста и может приводить к недействительности кешей ЦП, которые стали очень важными в обеспечении производительности современных процессоров.

Среда CLR избегает превышения лимита в пуле потоков за счет постановки задач в очередь и настройки их запуска. Она начинает с выполнения такого количества параллельных задач, которое соответствует числу аппаратных ядер, и затем регулирует уровень параллелизма по алгоритму поиска экстремума, непрерывно настраивая рабочую нагрузку в определенном направлении. Если производительность улучшается, тогда среда CLR продолжает двигаться в том же направлении (а иначе — в противоположном). В результате обеспечивается продвижение по оптимальной кривой производительности даже при наличии соперничающих процессов на компьютере.

Стратегия, реализованная в CLR, работает хорошо в случае удовлетворения следующих двух условий:

- элементы работы являются в основном кратковременными (менее 250 миллисекунд либо в идеале менее 100 миллисекунд), так что CLR имеет много возможностей для измерения и корректировки;
- в пуле не доминируют задания, которые большую часть своего времени являются заблокированными.

Блокирование ненадежно, поскольку дает среде CLR ложное представление о том, что оно загружает ЦП. Среда CLR достаточно интеллектуальна, чтобы обнаружить это и скомпенсировать (за счет внедрения дополнительных потоков в пул), хотя такое действие может сделать пул уязвимым к последующему превышению лимита. Также может быть введена задержка, потому что среда CLR регулирует скорость внедрения новых потоков, особенно на раннем этапе времени жизни приложения (тем более в клиентских ОС, где она отдает предпочтение низкому потреблению ресурсов).

Поддержание чистоты пула потоков особенно важно, когда требуется в полной мере задействовать ЦП (например, через API-интерфейсы параллельного программирования, рассматриваемые в главе 23).

# Задачи

Поток — это низкоуровневый инструмент для организации параллельной обработки и, будучи таковым, он обладает описанными ниже ограничениями.

- Несмотря на простоту передачи данных запускаемому потоку, не существует простого способа получить "возвращаемое значение" обратно из потока, для которого выполняется метод `Join`. Потребуется предусмотреть какое-то разделяемое поле. И если операция сгенерирует исключение, то его перехват и распространение будут сопряжены с аналогичными трудностями.
- После завершения потоку нельзя сообщить о том, что необходимо запустить что-нибудь еще; взамен к нему придется присоединяться с помощью метода `Join` (блокируя собственный поток в процессе).

Указанные ограничения препятствуют реализации мелкомодульного параллелизма; другими словами, они затрудняют формирование более крупных параллельных операций за счет комбинирования мелких операций (как будет показано в последующих разделах, это очень важно при асинхронном программировании). В свою очередь возникает более высокая зависимость от ручной синхронизации (блокировки, выдачи сигналов и т.д.) и проблем, которые ее сопровождают.

Прямое применение потоков также оказывает влияние на производительность, как обсуждалось ранее в разделе "Пул потоков". И если требуется запустить сотни или тысячи параллельных операций с интенсивным вводом-выводом, то подход на основе потоков повлечет за собой затраты сотен или тысяч мегабайтов памяти исключительно на накладные расходы, связанные с потоками.

Класс `Task`, реализующий задачу, помогает решить все упомянутые проблемы. В сравнении с потоком тип `Task` — абстракция более высокого уровня, т.к. он представляет параллельную операцию, которая может быть или не быть подкреплена потоком. Задачи поддерживают возможность *композиции* (их можно соединять вместе с использованием *продолжения*). Они могут работать с *пулом потоков* в целях снижения

задержки во время запуска, а с помощью класса TaskCompletionSource они позволяют задействовать подход с обратными вызовами, при котором потоки в целом не будут ожидать завершения операций с интенсивным вводом-выводом.

Типы Task появились в версии .NET Framework 4.0 как часть библиотеки параллельного программирования. Однако с тех пор они были усовершенствованы (за счет применения *объектов ожидания (awaiter)*), чтобы функционировать столь же эффективно в более универсальных сценариях реализации параллелизма, и имеют поддерживающие типы для асинхронных функций C#.

В настоящем разделе мы не затрагиваем возможности задач, предназначенные для параллельного программирования — они подробно рассматриваются в главе 23.

## Запуск задачи

Начиная с .NET Framework 4.5, простейший способ запуска задачи, подкрепленной потоком, предусматривает вызов статического метода Task.Run (класс Task находится в пространстве имен System.Threading.Tasks). Упомянутому методу нужно просто передать делегат Action:

```
Task.Run (() => Console.WriteLine ("Foo"));
```

Метод Task.Run был введен в .NET Framework 4.5. В версии .NET Framework 4.0 то же самое можно было сделать вызовом метода Task.Factory.StartNew. (Первый способ по большей части является сокращением для второго способа.)

По умолчанию задачи используют потоки из пула, которые являются фоновыми потоками. Это означает, что когда главный поток завершается, то завершаются и любые созданные вами задачи. Следовательно, чтобы запускать приводимые здесь примеры из консольного приложения, потребуется блокировать главный поток после старта задачи (скажем, ожидая завершения задачи или вызывая метод Console.ReadLine):

```
static void Main()
{
  Task.Run (() => Console.WriteLine ("Foo"));
  Console.ReadLine();
}
```

В сопровождающих книгу примерах для LINQPad вызов Console.ReadLine опущен, т.к. процесс LINQPad удерживает фоновые потоки в активном состоянии.

Вызов метода Task.Run в подобной манере похож на запуск потока следующим образом (за исключением влияния пула потоков, о котором речь пойдет чуть позже):

```
new Thread (() => Console.WriteLine ("Foo")).Start();
```

Метод Task.Run возвращает объект Task, который можно применять для мониторинга хода работ, почти как в случае объекта Thread. (Тем не менее, обратите внимание, что мы не вызываем метод Start после вызова Task.Run, т.к. метод Run создает "горячие" задачи; взамен можно воспользоваться конструктором класса Task и создавать "холодные" задачи, хотя на практике так поступают редко.)

Отслеживать состояние выполнения задачи можно с помощью ее свойства Status.

### Wait

Вызов метода Wait на объекте задачи приводит к блокированию до тех пор, пока она не будет завершена, и эквивалентен вызову метода Join на объекте потока:

```
Task task = Task.Run (() =>
{
  Thread.Sleep (2000);
  Console.WriteLine ("Foo");
});
Console.WriteLine (task.IsCompleted);  // False
task.Wait();                    // Блокируется вплоть до завершения задачи
```

Метод Wait позволяет дополнительно указывать тайм-аут и признак отмены для раннего завершения ожидания (см. раздел "Отмена" далее в главе).

### Длительно выполняющиеся задачи

По умолчанию среда CLR запускает задачи в потоках из пула, что идеально в случае кратковременных задач с интенсивными вычислениями. Для длительно выполняющихся и блокирующих операций (как в предыдущем примере) использование потоков из пула можно предотвратить, как показано ниже:

```
Task task = Task.Factory.StartNew (() => ...,
                                   TaskCreationOptions.LongRunning);
```

Запуск *одной* длительно выполняющейся задачи в потоке из пула не приведет к проблеме; производительность может пострадать, когда параллельно запускается несколько длительно выполняющихся задач (особенно таких, которые производят блокирование). И в этом случае обычно существуют более эффективные решения, нежели указание TaskCreationOptions.LongRunning:

- если задачи являются интенсивными в плане ввода-вывода, то вместо потоков следует применять класс TaskCompletionSource и асинхронные функции, которые позволяют реализовать параллельное выполнение с обратными вызовами (продолжениями);
- если задачи являются интенсивными в плане вычислений, то отрегулировать параллелизм для таких задач позволит очередь производителей/потребителей, избегая ограничения других потоков и процессов (см. раздел "Реализация очереди производителей/потребителей" в главе 23).

## Возвращение значений

Класс Task имеет обобщенный подкласс по имени Task<TResult>, который позволяет задаче выдавать возвращаемое значение. Для получения объекта Task<TResult> можно вызвать метод Task.Run с делегатом Func<TResult> (или совместимым лямбда-выражением) вместо делегата Action:

```
Task<int> task = Task.Run (() => { Console.WriteLine ("Foo"); return 3; });
// ...
```

Позже можно получить результат, запросив свойство Result.

Если задача еще не закончилась, то доступ к этому свойству заблокирует текущий поток до тех пор, пока задача не завершится:

```
int result = task.Result;    // Блокирует поток, если задача еще не завершена
Console.WriteLine (result); // 3
```

В следующем примере создается задача, которая использует LINQ для подсчета количества простых чисел в первых трех миллионах (+2) целочисленных значений:

```
Task<int> primeNumberTask = Task.Run (() =>
  Enumerable.Range (2, 3000000).Count (n =>
    Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

Console.WriteLine ("Task running...");
Console.WriteLine ("The answer is " + primeNumberTask.Result);
```

Код выводит строку `Task running...` (Задача выполняется...) и по прошествии нескольких секунд выдает ответ 216815.

Класс `Task<TResult>` можно воспринимать как "будущее" (future), поскольку он инкапсулирует свойство `Result`, которое станет доступным позже во времени.

Интересно отметить, что когда классы `Task` и `Task<TResult>` впервые появились в ранней предварительной версии (Community Technology Preview – CTP), то `Task<TResult>` действительно назывался `Future<TResult>`.

## Исключения

В отличие от потоков задачи без труда распространяют исключения. Таким образом, если код внутри задачи генерирует необработанное исключение (другими словами, если задача *отказывает*), то это исключение автоматически повторно сгенерируется при вызове метода `Wait` или доступе к свойству `Result` класса `Task<TResult>`:

```
// Запустить задачу, которая генерирует исключение NullReferenceException:
Task task = Task.Run (() => { throw null; });
try
{
  task.Wait();
}
catch (AggregateException aex)
{
  if (aex.InnerException is NullReferenceException)
    Console.WriteLine ("Null!");
  else
    throw;
}
```

(Среда CLR помещает исключение в оболочку `AggregateException` для нормальной работы в сценариях параллельного программирования; мы обсудим данный аспект в главе 23.)

Проверить, отказала ли задача, можно без повторной генерации исключения посредством свойств `IsFaulted` и `IsCanceled` класса `Task`. Если оба свойства возвращают `false`, то ошибки не возникали; если `IsCanceled` равно `true`, то для задачи было сгенерировано исключение `OperationCanceledException` (см. раздел "Отмена" далее в главе); если `IsFaulted` равно `true`, то было сгенерировано исключение другого типа и на ошибку укажет свойство `Exception`.

### Исключения и автономные задачи

В автономных задачах, работающих по принципу "установить и забыть" (для которых не требуется взаимодействие через метод `Wait` или свойство `Result` либо продолжение, делающее то же самое), общепринятой практикой является явное написание кода обработки исключений во избежание молчаливого отказа (в точности, как с потоком).

Необработанные исключения в автономных задачах называются *необнаруженными исключениями* и в CLR 4.0 они на самом деле завершают программу (среда CLR будет повторно генерировать исключение в потоке финализаторов, когда объект задачи покидает область видимости и обрабатывается сборщиком мусора). Это полезно для отражения факта возникновения проблемы, о которой вы могли быть не осведомлены; тем не менее, время появления ошибки иногда вводит в заблуждение из-за того, что сборщик мусора может существенно отставать от проблемной задачи. Следовательно, когда обнаружилось, что такое поведение усложняет некоторые шаблоны асинхронности (см. разделы "Параллелизм" и "`WhenAll`" далее в главе), в версии CLR 4.5 проблема была устранена.

Игнорирование исключений нормально в ситуации, когда исключение только указывает на неудачу при получении результата, который больше не интересует. Например, если пользователь отменяет запрос на загрузку веб-страницы, то мы не должны переживать, если выяснится, что веб-страница не существует.

Игнорирование исключений проблематично, когда исключение указывает на ошибку в программе, по двум причинам:

- ошибка может оставить программу в недопустимом состоянии;
- в результате ошибки позже могут возникнуть другие исключения, и отказ от регистрации первоначальной ошибки может затруднить диагностику.

Подписаться на необнаруженные исключения на глобальном уровне можно через статическое событие `TaskScheduler.UnobservedTaskException`; обработка этого события и регистрация ошибки нередко имеют смысл.

Есть пара интересных нюансов относительно того, какое исключение считать необнаруженным.

- Задачи, ожидающие с указанием тайм-аута, будут генерировать необнаруженное исключение, если ошибки возникают после истечения интервала тайм-аута.
- Действие по проверке свойства `Exception` задачи после ее отказа помечает исключение как обнаруженное.

## Продолжение

Продолжение сообщает задаче о том, что после завершения она должна продолжиться и делать что-то другое. Продолжение обычно реализуется посредством обратного вызова, который выполняется один раз после завершения операции. Существуют два способа присоединения признака продолжения к задаче. Первый из них был введен в .NET Framework 4.5 и особенно важен, поскольку применяется асинхронными функциями C#, что вскоре будет показано. Мы можем продемонстрировать его на примере с подсчетом простых чисел, который был реализован в разделе "Возвращение значений" ранее в главе:

```
Task<int> primeNumberTask = Task.Run (() =>
  Enumerable.Range (2, 3000000).Count (n =>
    Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));
var awaiter = primeNumberTask.GetAwaiter();
awaiter.OnCompleted (() =>
{
  int result = awaiter.GetResult();
  Console.WriteLine (result);  // Выводит значение result
});
```

Вызов метода `GetAwaiter` на объекте задачи возвращает объект *ожидания*, метод `OnCompleted` которого сообщает *предшествующей* задаче (`primeNumberTask`) о необходимости выполнить делегат, когда она завершится (или откажет). Признак продолжения допускается присоединять к уже завершенным задачам; в таком случае продолжение будет запланировано для немедленного выполнения.

Объект *ожидания* (awaiter) — это любой объект, открывающий доступ к двум методам, которые мы только что видели (`OnCompleted` и `GetResult`), и к булевскому свойству по имени `IsCompleted`. Никакого интерфейса или базового класса для унификации указанных членов не предусмотрено (хотя метод `OnCompleted` является частью интерфейса `INotifyCompletion`). Мы объясним важность данного шаблона в разделе "Асинхронные функции в C#" далее в главе.

Если предшествующая задача терпит отказ, то исключение генерируется повторно, когда код продолжения вызывает метод `awaiter.GetResult`. Вместо вызова `GetResult` мы могли бы просто обратиться к свойству `Result` предшествующей задачи. Преимущество вызова `GetResult` связано с тем, что в случае отказа предшествующей задачи исключение генерируется напрямую без помещения в оболочку `AggregateException`, позволяя писать более простые и чистые блоки `catch`.

Для необобщенных задач метод `GetResult` не имеет возвращаемого значения. Его польза состоит единственно в повторной генерации исключений.

Если присутствует контекст синхронизации, тогда метод `OnCompleted` его автоматически захватывает и отправляет ему признак продолжения. Это очень удобно в обогащенных клиентских приложениях, т.к. признак продолжения возвращается обратно потоку пользовательского интерфейса. Тем не менее, в случае библиотек подобное обычно нежелательно, потому что относительно затратный возврат в поток пользовательского интерфейса должен происходить только раз при покидании библиотеки, а не между вызовами методов. Следовательно, его можно аннулировать с помощью метода `ConfigureAwait`:

```
var awaiter = primeNumberTask.ConfigureAwait (false).GetAwaiter();
```

Когда контекст синхронизации отсутствует (или применяется `ConfigureAwait (false)`), продолжение будет (в общем случае) выполняться в том же самом потоке, что и предшествующий, избегая ненужных накладных расходов.

Другой способ присоединить продолжение предполагает вызов метода `ContinueWith` задачи:

```
primeNumberTask.ContinueWith (antecedent =>
{
  int result = antecedent.Result;
  Console.WriteLine (result);  // Выводит 123
});
```

Сам метод ContinueWith возвращает объект Task, который полезен, если планируется присоединение дальнейших признаков продолжения. Однако если задача отказывает, то в приложениях с пользовательским интерфейсом придется иметь дело напрямую с исключением AggregateException и предусмотреть дополнительный код для маршализации продолжения (см. раздел "Планировщики задач" в главе 23). В контекстах, не связанных с пользовательским интерфейсом, потребуется указывать TaskContinuationOptions.ExecuteSynchronously, если продолжение должно выполняться в том же потоке, иначе произойдет возврат в пул потоков. Метод ContinueWith особенно удобен в сценариях параллельного программирования; мы рассмотрим это подробно в разделе "Продолжение" главы 23.

## TaskCompletionSource

Ранее уже было указано, что метод Task.Run создает задачу, которая запускает делегат в потоке из пула (или не из пула). Еще один способ создания задачи заключается в использовании класса TaskCompletionSource.

Класс TaskCompletionSource позволяет создавать задачу из любой операции, которая начинается и через некоторое время заканчивается. Он работает путем предоставления "подчиненной" задачи, которой вы управляете вручную, указывая, когда операция завершилась или отказала. Это идеально для работы с интенсивным вводом-выводом: вы получаете все преимущества задач (с их возможностями передачи возвращаемых значений, исключений и признаков продолжения), не блокируя поток на период выполнения операции.

Для применения класса TaskCompletionSource нужно просто создать его экземпляр. Данный класс открывает доступ к свойству Task, возвращающему объект задачи, для которой можно организовать ожидание и присоединить признак продолжения — как делается с любой другой задачей. Тем не менее, такая задача полностью управляется объектом TaskCompletionSource с помощью следующих методов:

```
public class TaskCompletionSource<TResult>
{
  public void SetResult (TResult result);
  public void SetException (Exception exception);
  public void SetCanceled();
  public bool TrySetResult (TResult result);
  public bool TrySetException (Exception exception);
  public bool TrySetCanceled();
  public bool TrysetCanceled (CancellationToken cancellationToken);
  ...
}
```

Вызов одного из перечисленных методов *передает сигнал* задаче, помещая ее в состояние завершения, отказа или отмены (последнее состояние мы рассмотрим в разделе "Отмена" далее в главе). Предполагается, что вы будете вызывать любой из этих методов в точности один раз: в случае повторного вызова методы SetResult, SetException и SetCanceled сгенерируют исключение, а методы Try* возвратят false.

В следующем примере после пятисекундного ожидания выводится число 42:

```
var tcs = new TaskCompletionSource<int>();
new Thread (() => { Thread.Sleep (5000); tcs.SetResult (42); })
  { IsBackground = true }
  .Start();
```

```
Task<int> task = tcs.Task;                    // "Подчиненная" задача
Console.WriteLine (task.Result);              // 42
```

Можно реализовать собственный метод Run с использованием класса Task CompletionSource:

```
Task<TResult> Run<TResult> (Func<TResult> function)
{
  var tcs = new TaskCompletionSource<TResult>();
  new Thread (() =>
  {
    try { tcs.SetResult (function()); }
    catch (Exception ex) { tcs.SetException (ex); }
  }).Start();
  return tcs.Task;
}
...
Task<int> task = Run (() => { Thread.Sleep (5000); return 42; });
```

Вызов данного метода эквивалентен вызову Task.Factory.StartNew с параметром TaskCreationOptions.LongRunning для запроса потока не из пула.

Реальная мощь класса TaskCompletionSource состоит в возможности создания задач, не связывающих потоки. Например, рассмотрим задачу, которая ожидает пять секунд и затем возвращает число 42. Мы можем реализовать ее без потока с применением класса Timer, который с помощью CLR (и, в свою очередь, ОС) инициирует событие каждые *x* миллисекунд (таймеры еще будут рассматриваться в главе 22):

```
Task<int> GetAnswerToLife()
{
  var tcs = new TaskCompletionSource<int>();
  // Создать таймер, который инициирует событие раз в 5000 миллисекунд:
  var timer = new System.Timers.Timer (5000) { AutoReset = false };
  timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (42); };
  timer.Start();
  return tcs.Task;
}
```

Таким образом, наш метод возвращает объект задачи, которая завершается спустя пять секунд с результатом 42. Присоединив к задаче продолжение, мы можем вывести ее результат, не блокируя *ни одного* потока:

```
var awaiter = GetAnswerToLife().GetAwaiter();
awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult()));
```

Мы могли бы сделать код более полезным и превратить его в универсальный метод Delay, параметризировав время задержки и избавившись от возвращаемого значения. Это означало бы возвращение объекта Task вместо Task<int>. Тем не менее, необобщенной версии TaskCompletionSource не существует, а потому мы не можем напрямую создавать необобщенный объект Task. Обойти ограничение довольно просто: поскольку класс Task<TResult> является производным от Task, мы создаем TaskCompletionSource<*что-нибудь*> и затем неявно преобразуем получаемый Task<*что-нибудь*> в Task, примерно так:

```
var tcs = new TaskCompletionSource<object>();
Task task = tcs.Task;
```

Теперь можно реализовать универсальный метод Delay:

```
Task Delay (int milliseconds)
{
  var tcs = new TaskCompletionSource<object>();
  var timer = new System.Timers.Timer (milliseconds) { AutoReset = false };
  timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (null); };
  timer.Start();
  return tcs.Task;
}
```

Ниже показано, как использовать данный метод для вывода числа 42 после пятисекундной паузы:

```
Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

Такое применение класса `TaskCompletionSource` без потока означает, что поток будет занят, только когда запускается продолжение, т.е. спустя пять секунд. Мы можем продемонстрировать это, запустив 10 000 таких операций одновременно и не столкнувшись с ошибкой или чрезмерным потреблением ресурсов:

```
for (int i = 0; i < 10000; i++)
  Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

Таймеры инициируют свои обратные вызовы на потоках из пула, так что через пять секунд пул потоков получит 10 000 запросов вызова `SetResult(null)` на `TaskCompletionSource`. Если запросы поступают быстрее, чем они могут быть обработаны, тогда пул потоков отреагирует постановкой их в очередь и последующей обработкой на оптимальном уровне параллелизма для ЦП. Это идеально в ситуации, когда привязанные к потокам задания являются кратковременными, что в данном случае справедливо: привязанное к потоку задание просто вызывает метод `SetResult` и либо осуществляет отправку признака продолжения контексту синхронизации (в приложении с пользовательским интерфейсом), либо выполняет само продолжение (`Console.WriteLine(42)`).

## Task.Delay

Только что реализованный метод `Delay` достаточно полезен своей доступностью в качестве статического метода класса `Task`:

```
Task.Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

или:

```
Task.Delay (5000).ContinueWith (ant => Console.WriteLine (42));
```

Метод `Task.Delay` является *асинхронным* эквивалентом метода `Thread.Sleep`.

# Принципы асинхронности

Мы завершили демонстрацию `TaskCompletionSource` написанием *асинхронных* методов. В данном разделе мы объясним, что собой представляют асинхронные операции, и покажем, как они приводят к асинхронному программированию.

## Сравнение синхронных и асинхронных операций

*Синхронная операция* выполняет свою работу *перед* возвратом управления вызывающему коду.

*Асинхронная операция* выполняет (большую часть или всю) свою работу *после* возврата управления вызывающему коду.

Большинство создаваемых и вызываемых вами методов будут синхронными. Примерами могут служить `List<T>.Add`, `Console.WriteLine` и `Thread.Sleep`. Асинхронные методы менее распространены и инициируют *параллелизм*, т.к. они продолжают работать параллельно с вызывающим кодом. Асинхронные методы обычно быстро (или немедленно) возвращают управление вызывающему компоненту, потому их также называют *неблокирующими методами*.

Большинство асинхронных методов, которые мы видели до сих пор, могут быть описаны как универсальные методы:

- `Thread.Start`;
- `Task.Run`;
- методы, которые присоединяют признаки продолжения к задачам.

Вдобавок некоторые методы из числа рассмотренных в разделе "Контексты синхронизации" (`Dispatcher.BeginInvoke`, `Control.BeginInvoke` и `SynchronizationContext.Post`) являются асинхронными, как и методы, которые были написаны в разделе "`TaskCompletionSource`" ранее в главе, включая `Delay`.

## Что собой представляет асинхронное программирование?

Принцип асинхронного программирования состоит в том, что длительно выполняющиеся (или потенциально длительно выполняющиеся) функции реализуются асинхронным образом. Он отличается от традиционного подхода синхронной реализации длительно выполняющихся функций с последующим их вызовом в новом потоке или в задаче для введения параллелизма по мере необходимости.

Отличие от синхронного подхода в том, что параллелизм инициируется *внутри* длительно выполняющейся функции, а не *за ее пределами*. В результате появляются два преимущества.

- Параллельное выполнение с интенсивным вводом-выводом может быть реализовано без связывания потоков (как было продемонстрировано в разделе "`TaskCompletionSource`"), улучшая показатели масштабируемости и эффективности.
- Обогащенные клиентские приложения в итоге содержат меньше кода в рабочих потоках, что упрощает достижение безопасности в отношении потоков.

В свою очередь это приводит к двум различающимся сценариям использования асинхронного программирования. Первый из них связан с написанием (обычно серверных) приложений, которые эффективно обрабатывают большой объем параллельных операций ввода-вывода. Проблемой здесь является не обеспечение *безопасности* к потокам (т.к. разделяемое состояние обычно минимально), а достижение *эффективности* потоков; в частности, отсутствие потребления одного потока на сетевой запрос. Следовательно, в таком контексте выигрыш от асинхронности получают только операции с интенсивным вводом-выводом.

Второй сценарий применения касается упрощения поддержки безопасности в отношении потоков внутри обогащенных клиентских приложений. Он особенно актуален с ростом размера программы, поскольку для борьбы со сложностью мы обычно проводим рефакторинг крупных методов в методы меньших размеров, получая в результате цепочки методов, которые вызывают друг друга (*графы вызовов*).

Если любая операция внутри традиционного графа *синхронных* вызовов является длительно выполняющейся, тогда мы должны запускать целый граф вызовов в рабочем потоке, чтобы обеспечить отзывчивость пользовательского интерфейса. Таким образом, мы в конечном итоге получаем единственную параллельную операцию, которая охватывает множество методов (*крупномодульный параллелизм*), что требует учета безопасности к потокам для каждого метода в графе.

В случае графа *асинхронных* вызовов мы не должны запускать поток до тех пор, пока это не станет действительно необходимым — как правило, в нижней части графа (или вообще не запускать поток для операций с интенсивным вводом-выводом). Все остальные методы могут выполняться полностью в потоке пользовательского интерфейса со значительно упрощенной поддержкой безопасности в отношении потоков. В результате получается *мелкомодульный параллелизм* — последовательность небольших параллельных операций, между которыми выполнение возвращается в поток пользовательского интерфейса.

Чтобы извлечь из этого выгоду, операции с интенсивным вводом-выводом и интенсивными вычислениями должны быть реализованы асинхронным образом; хорошее эмпирическое правило предусматривает асинхронную реализацию любой операции, выполнение которой может занять более 50 миллисекунд.

(Оборотная сторона заключается в том, что *чрезмерно* мелкомодульная асинхронность может нанести ущерб производительности, потому что с асинхронными операциями связаны определенные накладные расходы, как будет показано в разделе "Оптимизация" далее в главе.)

В настоящей главе мы сосредоточим внимание главным образом на более сложном сценарии с обогащенным клиентом. В главе 16 будут приведены два примера, иллюстрирующие сценарий с интенсивным вводом-выводом (в разделах "Параллелизм и TCP" и "Реализация HTTP-сервера").

Инфраструктуры UWP (и Silverlight) поддерживают асинхронное программирование до момента, когда синхронные версии некоторых длительно выполняющихся методов либо не доступны, либо генерируют исключения. Взамен потребуется вызывать асинхронные методы, возвращающие объекты задач (или объекты, которые могут быть преобразованы в задачи посредством расширяющего метода `AsTask`).

## Асинхронное программирование и продолжение

Задачи идеально подходят для асинхронного программирования, т.к. они поддерживают признаки продолжения, которые являются жизненно важными в реализации асинхронности (взгляните на метод `Delay`, реализованный в разделе "`TaskCompletionSource`" ранее в главе). При написании метода `Delay` мы использовали класс `TaskCompletionSource`, который предлагает стандартный способ реализации асинхронных методов с интенсивным вводом-выводом "нижнего уровня".

В случае методов с интенсивными вычислениями для инициирования параллелизма, связанного с потоками, мы применяем метод `Task.Run`. Асинхронный метод создается просто за счет возвращения вызывающему компоненту объекта задачи. Асинхронное программирование отличается тем, что мы стремимся поступать подобным образом на как можно более низком уровне графа вызовов. Тогда высокоу-

ровневые методы в обогащенных клиентских приложениях могут быть оставлены в потоке пользовательского интерфейса и получать доступ к элементам управления и разделяемому состоянию, не порождая проблем с безопасностью к потокам. В целях иллюстрации рассмотрим показанный ниже метод, который вычисляет и подсчитывает простые числа, используя все доступные ядра (класс `ParallelEnumerable` обсуждается в главе 23):

```
int GetPrimesCount (int start, int count)
{
  return
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0));
}
```

Детали того, как работает алгоритм, не особенно важны; имеет значение лишь то, что метод требует некоторого времени на выполнение. Мы можем продемонстрировать это, написав другой метод, который вызывает `GetPrimesCount`:

```
void DisplayPrimeCounts()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (GetPrimesCount (i*1000000 + 2, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
  Console.WriteLine ("Done!");
}
```

Вот как выглядит вывод:

```
78498 primes between 0 and 999999
70435 primes between 1000000 and 1999999
67883 primes between 2000000 and 2999999
66330 primes between 3000000 and 3999999
65367 primes between 4000000 and 4999999
64336 primes between 5000000 and 5999999
63799 primes between 6000000 and 6999999
63129 primes between 7000000 and 7999999
62712 primes between 8000000 and 8999999
62090 primes between 9000000 and 9999999
```

Теперь у нас есть *граф вызовов* с методом `DisplayPrimeCounts`, обращающимся к методу `GetPrimesCount`. Для простоты внутри `DisplayPrimeCounts` применяется метод `Console.WriteLine`, хотя в реальности, скорее всего, будут обновляться элементы управления пользовательского интерфейса в обогащенном клиентском приложении, что демонстрируется позже. Крупномодульный параллелизм для такого графа вызовов можно инициировать следующим образом:

```
Task.Run (() => DisplayPrimeCounts());
```

В случае асинхронного подхода с мелкомодульным параллелизмом мы начинаем с написания асинхронной версии метода `GetPrimesCount`:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

## Важность языковой поддержки

Теперь мы должны модифицировать метод `DisplayPrimeCounts` так, чтобы он вызывал `GetPrimesCountAsync`. Именно здесь в игру вступают ключевые слова `await` и `async` языка C#, поскольку поступить по-другому намного сложнее, чем может показаться. Если мы просто изменим цикл, как показано ниже:

```
for (int i = 0; i < 10; i++)
{
  var awaiter = GetPrimesCountAsync (i*1000000 + 2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
    Console.WriteLine (awaiter.GetResult() + " primes between... "));
}
Console.WriteLine ("Done");
```

то цикл быстро пройдет через десять итераций (методы не являются блокирующими) и все десять операций будут выполняться параллельно (с ранним выводом строки `Done`).

Выполнять приведенные задачи параллельно в данном случае нежелательно, т.к. их внутренние реализации уже распараллелены; это приведет лишь к более длительному ожиданию первых результатов (и нарушению упорядочения).

Однако существует намного более распространенная причина для *последовательного* выполнения задач – ситуация, когда задача Б зависит от результатов выполнения задачи А. Например, при выборке веб-страницы DNS-поиск должен предшествовать HTTP-запросу.

Чтобы обеспечить последовательное выполнение, потребуется запускать следующую итерацию цикла из самого продолжения, что означает устранение цикла `for` и реализацию рекурсивного вызова в продолжении:

```
void DisplayPrimeCounts()
{
  DisplayPrimeCountsFrom (0);
}
void DisplayPrimeCountsFrom (int i)
{
  var awaiter = GetPrimesCountAsync (i*1000000 + 2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    Console.WriteLine (awaiter.GetResult() + " primes between...");
    if (i++ < 10) DisplayPrimeCountsFrom (i);
    else Console.WriteLine ("Done");
  });
}
```

Все становится еще хуже, если необходимо сделать асинхронным *сам* метод `DisplayPrimesCount`, возвращая объект задачи, которая отправляет сигнал о своем завершении. Достижение такой цели требует создания объекта `TaskCompletionSource`:

```
Task DisplayPrimeCountsAsync()
{
  var machine = new PrimesStateMachine();
  machine.DisplayPrimeCountsFrom (0);
  return machine.Task;
}
```

```
class PrimesStateMachine
{
  TaskCompletionSource<object> _tcs = new TaskCompletionSource<object>();
  public Task Task { get { return _tcs.Task; } }

public void DisplayPrimeCountsFrom (int i)
  {
    var awaiter = GetPrimesCountAsync (i*1000000+2, 1000000).GetAwaiter();
    awaiter.OnCompleted (() =>
    {
      Console.WriteLine (awaiter.GetResult());
      if (i++ < 10) DisplayPrimeCountsFrom (i);
      else { Console.WriteLine ("Done"); _tcs.SetResult (null); }
    });
  }
}
```

К счастью, *асинхронные функции* C# делают всю работу подобного рода. Благодаря новым ключевым словам `async` и `await` нам придется написать только следующий код:

```
async Task DisplayPrimeCountsAsync()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (await GetPrimesCountAsync (i*1000000 + 2, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
  Console.WriteLine ("Done!");
}
```

Таким образом, ключевые слова `async` и `await` очень важны для реализации асинхронности без чрезмерной сложности. Давайте посмотрим, как они работают.

Взглянуть на данную проблему можно и по-другому: императивные конструкции циклов (`for`, `foreach` и т.д.) не очень хорошо сочетаются с признаками продолжения, поскольку они полагаются на *текущее локальное состояние* метода (т.е. сколько раз цикл планирует выполняться).

Хотя ключевые слова `async` и `await` предлагают одно решение, иногда решить проблему удается другим способом, заменяя императивные конструкции циклов их *функциональными* эквивалентами (другими словами, запросами LINQ). Это является основой инфраструктуры *Reactive Framework* (Rx) и может оказаться удачным вариантом, когда в отношении результата нужно выполнить операции запросов или скомбинировать несколько последовательностей. Недостаток связан с тем, что во избежание блокировки инфраструктура Rx оперирует на последовательностях с *активным* источником, которые могут оказаться концептуально сложными.

# Асинхронные функции в C#

В версии C# 5.0 появились ключевые слова `async` и `await`. Они позволяют писать асинхронный код, обладающий той же самой структурой и простотой, что и синхронный код, а также устранять необходимость во вспомогательном коде, который присущ асинхронному программированию.

## Ожидание

Ключевое слово `await` упрощает присоединение признаков продолжения. Рассмотрим базовый сценарий. Приведенные ниже строки:

```
var результат = await выражение;
оператор(ы);
```

компилятор развернет в следующий функциональный эквивалент:

```
var awaiter = выражение.GetAwaiter();
awaiter.OnCompleted (() =>
{
  var результат = awaiter.GetResult();
  оператор(ы);
);
```

Компилятор также выпускает код для замыкания продолжения в случае синхронного завершения (см. раздел "Оптимизация" далее в главе) и для обработки разнообразных нюансов, которые мы затронем в последующих разделах.

В целях демонстрации вернемся к ранее написанному асинхронному методу, который вычисляет и подсчитывает простые числа:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

Используя ключевое слово `await`, его можно вызвать следующим образом:

```
int result = await GetPrimesCountAsync (2, 1000000);
Console.WriteLine (result);
```

Чтобы код скомпилировался, к содержащему такой вызов методу понадобится добавить модификатор `async`:

```
async void DisplayPrimesCount()
{
  int result = await GetPrimesCountAsync (2, 1000000);
  Console.WriteLine (result);
}
```

Модификатор `async` сообщает компилятору о необходимости трактовать `await` как ключевое слово, а не идентификатор, что привело бы к неоднозначности внутри данного метода (это гарантирует успешную компиляцию кода, написанного до выхода версии C# 5.0, где слово `await` использовалось в качестве идентификатора). Модификатор `async` может применяться только к методам (и лямбда-выражениям), которые возвращают `void` либо (как позже будет показано) тип `Task` или `Task<TResult>`.

Модификатор `async` подобен модификатору `unsafe` в том, что он не дает никакого эффекта на сигнатуре или открытых метаданных метода, а воздействует, только когда находится *внутри* метода. По этой причине не имеет смысла использовать `async` в интерфейсе. Однако вполне законно, например, вводить `async` при переопределении виртуального метода, не являющегося асинхронным, при условии сохранения сигнатуры метода в неизменном виде.

Методы с модификатором async называются *асинхронными функциями*, т.к. сами они обычно асинхронны. Чтобы увидеть почему, давайте посмотрим, каким образом процесс выполнения проходит через асинхронную функцию.

Встретив выражение await, процесс выполнения (обычно) производит возврат в вызывающий код – почти как оператор yield return в итераторе. Но перед возвратом исполняющая среда присоединяет к ожидающей задаче признак продолжения, который гарантирует, что когда задача завершится, управление перейдет обратно в метод и продолжит с места, где оно его оставило. Если задача отказывает, тогда ее исключение генерируется повторно, а в противном случае выражению await присваивается возвращаемое значение задачи. Все сказанное можно резюмировать, просмотрев логическое расширение показанного выше асинхронного метода:

```
void DisplayPrimesCount()
{
  var awaiter = GetPrimesCountAsync (2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    int result = awaiter.GetResult();
    Console.WriteLine (result);
  });
}
```

Выражение, к которому применяется await, обычно является задачей; тем не менее, компилятор устроит любой объект с методом GetAwaiter, который возвращает *объект с возможностью ожидания* (реализующий метод INotifyCompletion.OnCompleted и имеющий должным образом типизированный метод GetResult и свойство boolIsCompleted).

Обратите внимание, что выражение await оценивается как относящееся к типу int; причина в том, что ожидаемым выражением было Task<int> (метод GetAwaiter().GetResult которого возвращает тип int).

Ожидание необобщенной задачи вполне законно и генерирует выражение void:

```
await Task.Delay (5000);
Console.WriteLine ("Five seconds passed!");
```

### Захват локального состояния

Реальная мощь выражений await заключается в том, что они могут находиться практически в любом месте кода. В частности, выражение await может присутствовать на месте любого выражения (внутри асинхронной функции) кроме выражения lock, контекста unsafe или точки входа в исполняемый модуль (метод Main).

В следующем примере await располагается внутри цикла:

```
async void DisplayPrimeCounts()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (await GetPrimesCountAsync (i*1000000+2, 1000000));
}
```

При первом выполнении метода GetPrimesCountAsync управление возвращается вызывающему коду из-за выражения await. Когда метод завершается (или отказывает), выполнение возобновляется с места, в котором оно его покинуло, с сохраненными значениями локальных переменных и счетчиков циклов.

В отсутствие ключевого слова await простейшим эквивалентом мог бы служить пример, реализованный в разделе "Важность языковой поддержки" ранее в главе.

Однако компилятор реализует более общую стратегию преобразования таких методов в конечные автоматы (очень похоже на то, как он поступает с итераторами).

В возобновлении выполнения после выражения `await` компилятор полагается на признаки продолжения (согласно шаблону объектов ожидания). Это значит, что в случае запуска в потоке пользовательского интерфейса контекст синхронизации гарантирует, что выполнение будет возобновлено в том же самом потоке. В противном случае выполнение возобновляется в любом потоке, где задача была завершена. Смена потока не оказывает влияния на порядок выполнения и несущественна, если только вы каким-то образом не зависите от родства потоков, возможно, из-за использования локального хранилища потока (см. раздел "Локальное хранилище потока" в главе 22). Здесь уместна аналогия с ситуацией, когда вы ловите такси, чтобы добраться из одного места в другое. При наличии контекста синхронизации в вашем распоряжении всегда будет один и тот же таксомотор, а без контекста синхронизации таксомоторы каждый раз, возможно, окажутся разными. Хотя путешествие в любом случае будет в основном тем же самым.

### Ожидание в пользовательском интерфейсе

Мы можем продемонстрировать асинхронные функции в более практичном контексте, реализовав простой пользовательский интерфейс, который остается отзывчивым во время вызова метода с интенсивными вычислениями. Давайте начнем с синхронного решения:

```
class TestUI : Window
{
  Button _button = new Button { Content = "Go" };
  TextBlock _results = new TextBlock();
  public TestUI()
  {
    var panel = new StackPanel();
    panel.Children.Add (_button);
    panel.Children.Add (_results);
    Content = panel;
    _button.Click += (sender, args) => Go();
  }
  void Go()
  {
    for (int i = 1; i < 5; i++)
      _results.Text += GetPrimesCount (i * 1000000, 1000000) +
        " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
        Environment.NewLine;
  }
  int GetPrimesCount (int start, int count)
  {
    return ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0));
  }
}
```

После щелчка на кнопке Go (Запуск) приложение перестает быть отзывчивым на время, необходимое для выполнения кода с интенсивными вычислениями. Решение превращается в асинхронное за два шага. Первый шаг связан с переключением на асинхронную версию метода `GetPrimesCount`, который применялся в предыдущем примере:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

Второй шаг предусматривает изменение метода Go для вызова метода GetPrimesCountAsync:

```
async void Go()
{
  _button.IsEnabled = false;
  for (int i = 1; i < 5; i++)
    _results.Text += await GetPrimesCountAsync (i * 1000000, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
      Environment.NewLine;
  _button.IsEnabled = true;
}
```

Приведенный выше код демонстрирует простоту программирования с использованием асинхронных функций: все делается как при синхронном подходе, но вместо блокирования функций и их ожидания посредством await производится вызов асинхронных функций. В рабочем потоке запускается только код внутри метода GetPrimesCountAsync; код в методе Go "арендует" время у потока пользовательского интерфейса. Можно было бы сказать, что метод Go выполняется *псевдопараллельно* циклу сообщений (т.е. его выполнение пересекается с другими событиями, которые обрабатывает поток пользовательского интерфейса). Благодаря такому псевдопараллелизму единственной точкой, где может возникнуть вытеснение, является выполнение await. В итоге обеспечение безопасности к потокам упрощается: в данном случае может возникнуть только одна проблема – *реентерабельность* (из-за повторного щелчка на кнопке во время выполнения метода Go, чего мы избегаем, делая кнопку недоступной). Подлинный параллелизм происходит ниже в стеке вызовов, внутри кода, вызываемого методом Task.Run. Чтобы извлечь преимущества из такой модели, по-настоящему параллельный код избегает доступа к разделяемому состоянию или элементам управления пользовательского интерфейса.

Рассмотрим еще один пример, в котором вместо вычисления простых чисел загружается несколько веб-страниц с суммированием их длин. В .NET Framework 4.5 (и более поздних версиях) доступно множество асинхронных методов, возвращающих задачи, один из которых определен в классе WebClient внутри пространства имен System.Net. Метод DownloadDataTaskAsync асинхронно загружает URI в байтовый массив, возвращая объект Task<byte[]>, так что в результате ожидания можно получить массив byte[]. Давайте перепишем метод Go:

```
async void Go()
{
  _button.IsEnabled = false;
  string[] urls = "www.albahari.com www.oreilly.com www.linqpad.net".Split();
  int totalLength = 0;
  try
  {
    foreach (string url in urls)
    {
      var uri = new Uri ("http://" + url);
      byte[] data = await new WebClient().DownloadDataTaskAsync (uri);
```

```
      _results.Text += "Length of " + url + " is " + data.Length +
                       Environment.NewLine;
      totalLength += data.Length;
    }
    _results.Text += "Total length: " + totalLength;  // общая длина
  }
  catch (WebException ex)
  {
    _results.Text += "Error: " + ex.Message;
  }
  finally { _button.IsEnabled = true; }
}
```

И снова код отражает то, как бы мы реализовали его синхронным образом, включая применение блоков `catch` и `finally`. Хотя после первого `await` управление возвращается вызывающему коду, блок `finally` не выполняется вплоть до логического завершения метода (после выполнения всего его кода либо по причине раннего возвращения из-за оператора `return` или возникновения необработанного исключения).

Полезно посмотреть, что в точности происходит. Для начала необходимо вернуться к псевдокоду, который выполняет цикл сообщений в потоке пользовательского интерфейса:

```
Установить для этого потока контекст синхронизации WPF
while (приложение не завершено)
{
  Ожидать появления чего-нибудь в очереди сообщений
  Что-то получено: к какому виду сообщений оно относится?
    Сообщение клавиатуры/мыши -> запустить обработчик событий
    Пользовательское сообщение BeginInvoke/Invoke -> выполнить делегат
}
```

Обработчики событий, присоединяемые к элементам пользовательского интерфейса, выполняются через такой цикл сообщений. Когда запускается наш метод `Go`, выполнение продолжается до выражения `await`, после чего управление возвращается в цикл сообщений (освобождая пользовательский интерфейс для реагирования на дальнейшие события). Однако расширение компилятором выражения `await` гарантирует, что перед возвращением продолжение настроено так, чтобы выполнение возобновлялось там, где оно было прекращено до завершения задачи. И поскольку ожидание с помощью `await` происходит в потоке пользовательского интерфейса, то признак продолжения отправляется контексту синхронизации, который выполняет его через цикл сообщений, сохраняя выполнение всего метода `Go` псевдопараллельным с потоком пользовательского интерфейса. Подлинный параллелизм (с интенсивным вводом-выводом) происходит внутри реализации метода `DownloadDataTaskAsync`.

## Сравнение с крупномодульным параллелизмом

До выхода версии C# 5.0 асинхронное программирование было затруднено не только из-за отсутствия языковой поддержки, но и потому, что асинхронная функциональность в .NET Framework открывалась через неуклюжие шаблоны EAP и APM (см. раздел "Устаревшие шаблоны" далее в главе), а не через методы, возвращающие объекты задач.

Популярным обходным путем являлся крупномодульный параллелизм (для чего даже был предусмотрен тип по имени `BackgroundWorker`). Мы можем продемонстрировать крупномодульную асинхронность на исходном *синхронном* примере с методом `GetPrimesCount`, изменив обработчик событий кнопки, как показано ниже:

```
...
_button.Click += (sender, args) =>
{
  _button.IsEnabled = false;
  Task.Run (() => Go());
};
```

(Мы решили использовать метод `Task.Run` вместо класса `BackgroundWorker` из-за того, что `BackgroundWorker` никак бы не упростил данный конкретный пример.) В любом случае конечный результат состоит в том, что целый граф синхронных вызовов (`Go` и `GetPrimesCount`) выполняется в рабочем потоке. И поскольку метод `Go` обновляет элементы пользовательского интерфейса, в код придется добавить вызовы `Dispatcher.BeginInvoke`:

```
void Go()
{
  for (int i = 1; i < 5; i++)
  {
    int result = GetPrimesCount (i * 1000000, 1000000);
    Dispatcher.BeginInvoke (new Action (() =>
      _results.Text += result + " primes between " + (i*1000000) +
      " and " + ((i+1)*1000000-1) + Environment.NewLine));
  }
  Dispatcher.BeginInvoke (new Action (() => _button.IsEnabled = true));
}
```

В отличие от асинхронной версии цикл сам выполняется в рабочем потоке. Это может казаться безобидным, но даже в таком простом случае применение многопоточности привело к возникновению условия состязаний. (Смогли его заметить? Если нет, тогда запустите программу: условие состязаний почти наверняка станет очевидным.)

Реализация отмены и сообщения о ходе работ создает больше возможностей для ошибок, связанных с нарушением безопасности к потокам, как делает любой дополнительный код в методе. Например, предположим, что верхний предел для цикла не закодирован жестко, а поступает из вызова метода:

```
for (int i = 1; i < GetUpperBound(); i++)
```

Далее представим, что метод `GetUpperBound` читает значение из конфигурационного файла, который ленивым образом загружается с диска при первом вызове. Весь этот код теперь выполняется в рабочем потоке — код, который весьма вероятно не является безопасным к потокам. В том и заключается опасность запуска рабочих потоков на высоких уровнях внутри графа вызовов.

## Написание асинхронных функций

Что касается любой асинхронной функции, то возвращаемый тип `void` можно заменить типом `Task`, чтобы сделать сам метод *пригодным* для асинхронного выполнения (и ожидания с помощью `await`). Никаких других изменений вносить не придется:

```
async Task PrintAnswerToLife()  // Вместо void можно возвращать Task
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  Console.WriteLine (answer);
}
```

Обратите внимание, что в теле метода мы не возвращаем объект задачи явным образом. Компилятор произведет задачу, которая будет отправлять сигнал о завершении данного метода (или о возникновении необработанного исключения). В итоге упрощается создание цепочек асинхронных вызовов:

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}
```

Так как метод `Go` объявлен с возвращаемым типом `Task`, сам `Go` допускает ожидание посредством `await`.

Компилятор расширяет асинхронные функции, возвращающие задачи, в код, использующий класс `TaskCompletionSource` для создания задачи, которая затем отправляет сигнал о завершении или отказе.

Компилятор фактически обращается к `TaskCompletionSource` косвенно, через типы `Async*MethodBuilder` из пространства имен `System.CompilerServices`. Такие типы обрабатывают крайние случаи вроде помещения задачи в состояние отмены при возникновении исключения `OperationCanceledException` и реализуют нюансы, описанные в разделе "Асинхронность и контексты синхронизации" далее в главе.

Оставив в стороне нюансы, мы можем развернуть метод `PrintAnswerToLife` в такой функциональный эквивалент:

```
Task PrintAnswerToLife()
{
  var tcs = new TaskCompletionSource<object>();
  var awaiter = Task.Delay (5000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    try
    {
      awaiter.GetResult();  // Сгенерировать повторно любые исключения
      int answer = 21 * 2;
      Console.WriteLine (answer);
      tcs.SetResult (null);
    }
    catch (Exception ex) { tcs.SetException (ex); }
  });
  return tcs.Task;
}
```

Следовательно, всякий раз, когда возвращающий задачу асинхронный метод завершается, управление переходит обратно к месту его ожидания (благодаря признаку продолжения).

В сценарии с обогащенным клиентом управление перемещается с этой точки обратно в поток пользовательского интерфейса (если оно еще в нем не находится). В противном случае выполнение продолжается в любом потоке, куда был направлен признак продолжения, что означает отсутствие задержки при подъеме по графу асинхронных вызовов кроме первого "прыжка", если он был инициирован потоком пользовательского интерфейса.

### Возврат Task<TResult>

Возвращать объект Task<TResult> можно, если в теле метода возвращается тип TResult:

```
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;      // Метод имеет возвращаемый тип Task<int>,
                      // поэтому возвратить int
}
```

Внутренне это приводит к тому, что объект TaskCompletionSource отправляется сигнал со значением, отличающимся от null. Мы можем продемонстрировать работу метода GetAnswerToLife, вызвав его из метода PrintAnswerToLife (который в свою очередь вызывается из Go):

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}
async Task PrintAnswerToLife()
{
  int answer = await GetAnswerToLife();
  Console.WriteLine (answer);
}
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;
}
```

В сущности, мы преобразовали исходный метод PrintAnswerToLife в два метода — с той же легкостью, как если бы программировали синхронным образом. Сходство с синхронным программированием является умышленным; вот синхронный эквивалент нашего графа вызовов, для которого вызов метода Go дает тот же самый результат после блокирования на протяжении пяти секунд:

```
void Go()
{
  PrintAnswerToLife();
  Console.WriteLine ("Done");
}
void PrintAnswerToLife()
{
  int answer = GetAnswerToLife();
  Console.WriteLine (answer);
}
int GetAnswerToLife()
{
  Thread.Sleep (5000);
  int answer = 21 * 2;
  return answer;
}
```

Тем самым также иллюстрируется базовый принцип проектирования с применением асинхронных функций в C#.

1. Напишите синхронные версии своих методов.
2. Замените вызовы *синхронных* методов вызовами *асинхронных* методов и примените к ним `await`.
3. За исключением методов "верхнего уровня" (обычно обработчиков событий для элементов управления пользовательского интерфейса) поменяйте возвращаемые типы асинхронных методов на `Task` или `Task<TResult>`, чтобы они поддерживали `await`.

Способность компилятора производить задачи для асинхронных функций означает, что явно создавать объект `TaskCompletionSource` придется главным образом только в методах нижнего уровня, которые инициируют параллелизм с интенсивным вводом-выводом. (Для методов, инициирующих параллелизм с интенсивными вычислениями, создается задача с помощью метода `Task.Run`.)

### Выполнение графа асинхронных вызовов

Чтобы ясно увидеть, как все выполняется, полезно реорганизовать код следующим образом:

```
async Task Go()
{
  var task = PrintAnswerToLife();
  await task; Console.WriteLine ("Done");
}
async Task PrintAnswerToLife()
{
  var task = GetAnswerToLife();
  int answer = await task; Console.WriteLine (answer);
}
async Task<int> GetAnswerToLife()
{
  var task = Task.Delay (5000);
  await task; int answer = 21 * 2; return answer;
}
```

Метод `Go` вызывает метод `PrintAnswerToLife`, который вызывает метод `GetAnswerToLife`, а тот в свою очередь вызывает метод `Delay` и затем ожидает. Наличие `await` приводит к тому, что управление возвращается методу `PrintAnswerToLife`, который сам ожидает, возвращая управление методу `Go`, который также ожидает и возвращает управление вызывающему коду. Все происходит синхронно в потоке, вызвавшем метод `Go`; это короткая *синхронная* фаза выполнения.

Спустя пять секунд запускается продолжение на `Delay` и управление возвращается методу `GetAnswerToLife` в потоке из пула. (Если мы начинали в потоке пользовательского интерфейса, то управление возвратится в него.) Затем выполняются оставшиеся операторы в методе `GetAnswerToLife`, после чего задача `Task<int>` данного метода завершается с результатом 42 и инициируется продолжение в методе `PrintAnswerToLife`, что приведет к выполнению оставшихся операторов в этом методе. Процесс продолжается до тех пор, пока задача метода `Go` не выдаст сигнал о своем завершении.

Поток выполнения соответствует показанному ранее графу синхронных вызовов, т.к. мы следуем шаблону, при котором к каждому асинхронному методу сразу после вы-

зова применяется await. В результате создается последовательный поток без параллелизма или перекрывающегося выполнения внутри графа вызовов. Каждое выражение await образует "брешь" в выполнении, после которой программа возобновляет работу с того места, где она ее оставила.

### Параллелизм

Вызов асинхронного метода без его ожидания позволяет коду, который за ним следует, выполняться параллельно. В приведенных ранее примерах вы могли отметить наличие кнопки, обработчик события которой вызывал метод Go так, как показано ниже:

```
_button.Click += (sender, args) => Go();
```

Несмотря на то что Go является асинхронным методом, мы не можем применить к нему await, и это действительно то, что содействует параллелизму, необходимому для поддержки отзывчивого пользовательского интерфейса.

Тот же самый принцип можно использовать для запуска двух асинхронных операций параллельно:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

(За счет ожидания обеих операций впоследствии мы "заканчиваем" параллелизм в данной точке. Позже мы покажем, как комбинатор задач WhenAll помогает в реализации такого шаблона.)

Параллелизм, организованный подобным образом, происходит независимо от того, инициированы ли операции в потоке пользовательского интерфейса, хотя существует отличие в том, как он проявляется. В обоих случаях мы получаем тот же самый "подлинный" параллелизм в операциях нижнего уровня, которые его инициируют (таких как Task.Delay или код, предоставленный методу Task.Run). Методы, находящиеся выше в стеке вызовов, будут по-настоящему параллельными, только если операция была инициирована без наличия контекста синхронизации; иначе они окажутся псевдопараллельными (и упростят обеспечение безопасности к потокам), согласно чему единственным местом, где может произойти вытеснение, является оператор await. Это позволяет, например, определить разделяемое поле _x и инкрементировать его в методе GetAnswerToLife без блокирования:

```
async Task<int> GetAnswerToLife()
{
  _x++;
  await Task.Delay (5000);
  return 21 * 2;
}
```

(Тем не менее, мы не можем предполагать, что _x имеет одно и то же значение до и после await.)

## Асинхронные лямбда-выражения

Точно так же как обычные *именованные* методы могут быть асинхронными:

```
async Task NamedMethod()
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
}
```

асинхронными способны быть и *неименованные* методы (лямбда-выражения и анонимные методы), если они предварены ключевым словом async:

```
Func<Task> unnamed = async () =>
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
};
```

Вызывать и применять await к ним можно одинаково:

```
await NamedMethod();
await unnamed();
```

Асинхронные лямбда-выражения могут использоваться при подключении обработчиков событий:

```
myButton.Click += async (sender, args) =>
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

Это более лаконично, чем приведенный ниже код, который обеспечивает тот же самый результат:

```
myButton.Click += ButtonHandler;
...
async void ButtonHander (object sender, EventArgs args)
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

Асинхронные лямбда-выражения могут также возвращать тип Task<TResult>:

```
Func<Task<int>> unnamed = async () =>
{
  await Task.Delay (1000);
  return 123;
};
int answer = await unnamed();
```

## Асинхронные методы в WinRT

В WinRT эквивалентом типа Task является IAsyncAction, а эквивалентом Task<TResult> — тип IAsyncOperation<TResult> (из пространства имен Windows.Foundation).

Выполнять преобразование в тип Task или Task<TResult> либо из него можно с помощью расширяющего метода AsTask из сборки System.Runtime.WindowsRuntime.dll. В указанной сборке также определен метод GetAwaiter, оперирующий на типах IAsyncAction и IAsyncOperation<TResult>, которые позволяют реализовать ожидание напрямую. Например:

```
Task<StorageFile> fileTask = KnownFolders.DocumentsLibrary.CreateFileAsync
                             ("test.txt").AsTask();
```

или:

```
StorageFile file = await KnownFolders.DocumentsLibrary.CreateFileAsync
                         ("test.txt");
```

Из-за ограничений системы типов COM интерфейс `IAsyncOperation <TResult>` не основан на `IAsyncAction`, как можно было бы ожидать. Взамен оба интерфейса унаследованы от общего базового типа по имени `IAsyncInfo`.

Метод `AsTask` также перегружен для приема признака отмены (см. раздел "Отмена" далее в главе) и объекта `IProgress<T>` (см. раздел "Сообщение о ходе работ" также далее в главе).

## Асинхронность и контексты синхронизации

Ранее уже было показано, что наличие контекста синхронизации играет важную роль в смысле отправки признаков продолжения. В случае асинхронных функций, возвращающих `void`, существует пара других, более тонких способов взаимодействия с контекстами синхронизации. Они являются не прямым результатом расширений, производимых компилятором C#, а функцией типов `Async*MethodBuilder` из пространства имен `System.CompilerServices`, которое компилятор использует при расширении асинхронных функций.

### Отправка исключений

В обогащенных клиентских приложениях общепринято полагаться на событие централизованной обработки исключений (`Application.DispatcherUnhandled Exception` в WPF) для учета необработанных исключений, сгенерированных в потоке пользовательского интерфейса. В приложениях ASP.NET похожую работу делает метод `Application_Error` в *`global.asax`*. Внутренне они функционируют, инициируя события пользовательского интерфейса (или конвейера методов обработки страниц в случае ASP.NET) в собственном блоке `try/catch`. Асинхронные функции верхнего уровня затрудняют это. Рассмотрим следующий обработчик события щелчка на кнопке:

```
async void ButtonClick (object sender, RoutedEventArgs args)
{
  await Task.Delay(1000);
  throw new Exception ("Will this be ignored?"); //Будет ли это проигнорировано?
}
```

Когда на кнопке осуществляется щелчок и обработчик событий запускается, после оператора `await` управление обычно возвращается в цикл сообщений, и сгенерированное секунду спустя исключение не может быть перехвачено блоком `catch` в цикле сообщений.

Чтобы устранить проблему, структура `AsyncVoidMethodBuilder` перехватывает необработанные исключения (в асинхронных функциях, возвращающих `void`) и отправляет их контексту синхронизации, если он присутствует, гарантируя то, что события глобальной обработки исключений по-прежнему инициируются.

Компилятор применяет упомянутую логику только к асинхронным функциям, возвращающим *`void`*. Следовательно, если изменить `ButtonClick` для возвращения типа `Task` вместо `void`, тогда необработанное исключение приведет к отказу результирующей задачи, поскольку ему некуда больше двигаться (в результате давая *необнаруженное* исключение).

Интересный нюанс связан с тем, что нет никакой разницы, где генерируется исключение – до или после `await`. Таким образом, в следующем примере исключение отправляется контексту синхронизации (если он существует), но не вызывающему коду:

```
async void Foo() { throw null; await Task.Delay(1000); }
```

При отсутствии контекста синхронизации исключение станет необнаруженным. Может показаться странным, что исключение не возвращается обратно вызывающему коду, хотя ситуация не особо отличается от того, что происходит с итераторами:

```
IEnumerable<int> Foo() { throw null; yield return 123; }
```

В приведенном примере исключение никогда не возвращается прямо вызывающему коду: с исключением имеет дело только перечисляемая последовательность.

### OperationStarted и OperationCompleted

Если контекст синхронизации присутствует, то асинхронные функции, возвращающие `void`, также вызывают его метод `OperationStarted` при входе в функцию и метод `OperationCompleted`, когда функция завершается. Указанные методы используются контекстом синхронизации ASP.NET для обеспечения последовательного выполнения внутри конвейера обработки страниц.

Переопределение таких методов удобно при написании специального контекста синхронизации для проведения модульного тестирования асинхронных методов, возвращающих `void`. Данная тема обсуждается в блоге, посвященном параллельному программированию в .NET, по адресу `http://blogs.msdn.com/b/pfxteam`.

## Оптимизация

### Синхронное завершение

Возврат из асинхронной функции может произойти *перед* организацией ожидания. Рассмотрим следующий метод, который обеспечивает кеширование в процессе загрузки веб-страниц:

```
static Dictionary<string,string> _cache = new Dictionary<string,string>();
async Task<string> GetWebPageAsync (string uri)
{
  string html;
  if (_cache.TryGetValue (uri, out html)) return html;
  return _cache [uri] =
    await new WebClient().DownloadStringTaskAsync (uri);
}
```

Если URI уже присутствует в кеше, тогда управление возвращается вызывающему коду безо всякого ожидания, и метод возвращает объект задачи, которая *уже сигнализирована*. Это называется синхронным завершением.

Когда производится ожидание синхронно завершенной задачи, управление не возвращается вызывающему коду и не переходит обратно через признак продолжения — взамен выполнение продолжается со следующего оператора. Компилятор реализует такую оптимизацию, проверяя свойство `IsCompleted` объекта ожидания; другими словами, всякий раз, когда производится ожидание:

```
Console.WriteLine (await GetWebPageAsync ("http://oreilly.com"));
```

компилятор выпускает код для короткого замыкания продолжения в случае синхронного завершения:

```
var awaiter = GetWebPageAsync().GetAwaiter();
if (awaiter.IsCompleted)
  Console.WriteLine (awaiter.GetResult());
else
  awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult());
```

Ожидание асинхронной функции, которая завершается синхронно, все равно связано с небольшими накладными расходами – примерно 50-100 наносекунд на современных компьютерах.

Напротив, переход в пул потоков вызывает переключение контекста – возможно одну или две микросекунды, а переход в цикл сообщений пользовательского интерфейса – минимум в десять раз больше (и еще больше, если пользовательский интерфейс занят).

Вполне законно даже писать асинхронные методы, для которых *никогда* не производится ожидание, хотя компилятор сгенерирует предупреждение:

```
async Task<string> Foo() { return "abc"; }
```

Такие методы могут быть полезны при переопределении виртуальных/абстрактных методов, если случится так, что ваша реализация не потребует асинхронности. (Примером могут служить методы `ReadAsync`/`WriteAsync` класса `MemoryStream`, которые рассматриваются в главе 15.) Другой способ достичь того же результата предусматривает применение метода `Task.FromResult`, который возвращает уже сигнализированную задачу:

```
Task<string> Foo() { return Task.FromResult ("abc"); }
```

Если наш метод `GetWebPageAsync` вызывается из потока пользовательского интерфейса, то он является неявно безопасным к потокам в том, что его можно было бы вызвать несколько раз подряд (инициируя тем самым множество параллельных загрузок) без необходимости в каком-либо блокировании с целью защиты кеша. Однако если бы последовательность обращений относилась к одному и тому же URI, то инициировалось бы множество избыточных загрузок, которые все в конечном итоге обновляли бы одну и ту же запись в кеше (в выигрыше окажется последняя загрузка). Хоть это и не ошибка, но более эффективно было бы сделать так, чтобы последующие обращения к тому же самому URI взамен (асинхронно) ожидали результата выполняющегося запроса.

Существует простой способ достичь указанной цели, не прибегая к блокировкам или сигнализирующим конструкциям. Вместо кеша строк мы создаем кеш "будущего" (`Task<string>`):

```
static Dictionary<string,Task<string>> _cache =
  new Dictionary<string,Task<string>>();
Task<string> GetWebPageAsync (string uri)
{
  Task<string> downloadTask;
  if (_cache.TryGetValue (uri, out downloadTask)) return downloadTask;
  return _cache [uri] = new WebClient().DownloadStringTaskAsync (uri);
}
```

(Обратите внимание, что мы не помечаем метод как `async`, поскольку напрямую возвращаем объект задачи, полученный в результате вызова метода класса `WebClient`.)

Теперь при повторяющихся вызовах метода `GetWebPageAsync` с тем же самым URI мы гарантируем получение одного и того же объекта `Task<string>`. (Это обеспечивает и дополнительное преимущество минимизации нагрузки на сборщик мусора.) И если задача завершена, то ожидание не требует больших затрат благодаря описанной выше оптимизации, которую предпринимает компилятор.

Мы могли бы и дальше расширять пример, чтобы сделать его безопасным к потокам без защиты со стороны контекста синхронизации, для чего необходимо блокировать все тело метода:

```
lock (_cache)
{
  Task<string> downloadTask;
  if (_cache.TryGetValue (uri, out downloadTask)) return downloadTask;
  return _cache [uri] = new WebClient().DownloadStringTaskAsync (uri);
}
```

Код работает из-за того, что мы производим блокировку не на время загрузки страницы (это нанесло бы ущерб параллелизму), а на небольшой промежуток времени, пока проверяется кеш и при необходимости запускается новая задача, которая обновляет кеш.

### Избегание чрезмерных возвратов

В методах, которые многократно вызываются в цикле, можно избежать накладных расходов, связанных с повторяющимся возвратом в цикл сообщений пользовательского интерфейса, за счет вызова метода `ConfigureAwait`. В результате задача не передает признаки продолжения контексту синхронизации, сокращая накладные расходы до затрат на переключение контекста (или намного меньших, если метод, для которого осуществляется ожидание, завершается синхронно):

```
async void A() { ... await B(); ... }
async Task B()
{
  for (int i = 0; i < 1000; i++)
    await C().ConfigureAwait (false);
}
async Task C() { ... }
```

Это означает, что для методов `B` и `C` мы аннулируем простую модель безопасности к потокам в приложениях с пользовательским интерфейсом, согласно которой код выполняется в потоке пользовательского интерфейса и может быть вытеснен только во время выполнения оператора `await`. Однако метод `A` не затрагивается и останется в потоке пользовательского интерфейса, если он в нем был запущен. Такая оптимизация особенно уместна при написании библиотек: преимущество упрощенной безопасности потоков не требуется, потому что код библиотеки обычно не разделяет состояние с вызывающим кодом и не обращается к элементам управления пользовательского интерфейса. (В приведенном выше примере также имеет смысл реализовать синхронное завершение метода `C`, если известно, что операция, скорее всего, окажется кратковременной.)

# Асинхронные шаблоны

## Отмена

Часто важно иметь возможность отмены параллельной операции после ее запуска, скажем, в ответ на пользовательский запрос. Реализовать это проще всего с помощью флага отмены, который можно было бы инкапсулировать в классе следующего вида:

```
class CancellationToken
{
  public bool IsCancellationRequested { get; private set; }
  public void Cancel() { IsCancellationRequested = true; }
  public void ThrowIfCancellationRequested()
  {
    if (IsCancellationRequested)
      throw new OperationCanceledException();
  }
}
```

Затем можно было бы написать асинхронный метод с возможностью отмены:

```
async Task Foo (CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine (i);
    await Task.Delay (1000);
    cancellationToken.ThrowIfCancellationRequested();
  }
}
```

Когда вызывающий код желает отменить операцию, он обращается к методу `Cancel` признака отмены, который передается в метод `Foo`. В результате `IsCancellationRequested` устанавливается в `true`, что через краткий промежуток времени приводит к отказу метода `Foo` с генерацией исключения `OperationCanceledException` (предопределенный в пространстве имен `System` класс, который предназначен для данной цели).

Если оставить в стороне безопасность к потокам (мы должны блокировать чтение/запись в `IsCancellationRequested`), то такой шаблон вполне эффективен, и среда CLR предлагает тип по имени `CancellationToken`, который очень похож на только что рассмотренный тип. Тем не менее, в нем отсутствует метод `Cancel`; этот метод открыт в другом типе — `CancellationTokenSource`. Подобное разделение обеспечивает определенную безопасность: метод, который имеет доступ только к объекту `CancellationToken`, может проверять, но не *инициировать* отмену.

Чтобы получить признак отмены, сначала необходимо создать экземпляр `CancellationTokenSource`:

```
var cancelSource = new CancellationTokenSource();
```

После этого станет доступным свойство `Token`, которое возвращает объект `CancellationToken`. Таким образом, вызвать наш метод `Foo` можно было бы следующим образом:

```
var cancelSource = new CancellationTokenSource();
Task foo = Foo (cancelSource.Token);
...
... (в какой-то момент позже)
cancelSource.Cancel();
```

Признаки отмены поддерживает большинство асинхронных методов в CLR, включая `Delay`. Если модифицировать метод `Foo` так, чтобы он передавал свой признак отмены методу `Delay`, то задача будет завершаться немедленно по запросу (а не секунду спустя):

```
async Task Foo (CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine (i);
    await Task.Delay (1000, cancellationToken);
  }
}
```

Обратите внимание, что нам больше не понадобится вызывать метод `ThrowIfCancellationRequested`, поскольку это делает `Task.Delay`. Признаки отмены нормально распространяются вниз по стеку вызовов (так же как запросы отмены каскадным образом продвигаются *вверх* по стеку вызовов посредством исключений).

Асинхронные методы в WinRT при отмене следуют низкоуровневому протоколу, согласно которому вместо принятия `CancellationToken` тип `IAsyncInfo` открывает доступ к методу `Cancel`. Однако метод `AsTaskextension` перегружен для приема признака отмены, ликвидируя данный разрыв.

Синхронные методы также могут поддерживать отмену (как делает метод `Wait` класса `Task`). В таких случаях инструкция для отмены должна будет поступать асинхронно (скажем, из другой задачи). Например:

```
var cancelSource = new CancellationTokenSource();
Task.Delay (5000).ContinueWith (ant => cancelSource.Cancel());
...
```

В действительности, начиная с версии .NET Framework 4.5, при конструировании `CancellationTokenSource` можно указывать временной интервал, чтобы инициировать отмену по его прошествии (как только что было продемонстрировано). Прием удобен для реализации тайм-аутов, как синхронных, так и асинхронных:

```
var cancelSource = new CancellationTokenSource (5000);
try { await Foo (cancelSource.Token); }
catch (OperationCanceledException ex) { Console.WriteLine ("Cancelled"); }
```

Структура `CancellationToken` предоставляет метод `Register`, позволяющий зарегистрировать делегат обратного вызова, который будет запущен при отмене; он возвращает объект, который можно освободить с целью отмены регистрации.

Задачи, генерируемые асинхронными функциями компилятора, автоматически входят в состояние отмены при появлении необработанного исключения `OperationCanceledException` (свойство `IsCanceled` возвращает `true`, а свойство `IsFaulted` — `false`). То же самое происходит и в случае задач, созданных с помощью метода `Task.Run`, конструктору которых передается (тот же признак) `CancellationToken`. Отличие между отказавшей и отмененной задачей в асинхронных сценариях не является важным, т.к. обе они генерируют исключение `OperationCanceledException` во время ожидания; это играет роль в расширенных сценариях параллельного программирования (особенно при условном продолжении). Мы обсудим данную тему в разделе "Отмена задач" главы 23.

## Сообщение о ходе работ

Иногда нужно, чтобы асинхронная операция во время выполнения сообщала о ходе работ. Простое решение заключается в передаче асинхронному методу делегата `Action`, который запускается всякий раз, когда состояние хода работ меняется:

```
Task Foo (Action<int> onProgressPercentChanged)
{
  return Task.Run (() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0) onProgressPercentChanged (i / 10);
      // Делать что-нибудь, требующее интенсивных вычислений...
    }
  });
}
```

Вот как его можно вызывать:

```
Action<int> progress = i => Console.WriteLine (i + " %");
await Foo (progress);
```

Хотя такой прием нормально работает в консольном приложении, он не идеален в сценариях обогащенных клиентов, поскольку сообщает о ходе работ из рабочего потока, вызывая потенциальные проблемы с безопасностью к потокам у потребителя. (Фактически мы позволяем побочному эффекту от параллелизма "просочиться" во внешний мир, что нежелательно, т.к. в противном случае метод изолируется, если он вызван в потоке пользовательского интерфейса.)

### IProgress<T> и Progress<T>

Для решения описанной выше проблемы среда CLR предлагает пару типов: интерфейс `IProgress<T>` и класс `Progress<T>`, который реализует этот интерфейс. В действительности они предназначены для того, чтобы служить оболочкой делегата, позволяя приложениям с пользовательским интерфейсом безопасно сообщать о ходе работ через контекст синхронизации.

Интерфейс `IProgress<T>` определяет только один метод:

```
public interface IProgress<in T>
{
  void Report (T value);
}
```

Использовать интерфейс `IProgress<T>` легко; наш метод почти не изменяется:

```
Task Foo (IProgress<int> onProgressPercentChanged)
{
  return Task.Run (() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0) onProgressPercentChanged.Report (i / 10);
      // Делать что-нибудь, требующее интенсивных вычислений...
    }
  });
}
```

Класс `Progress<T>` имеет конструктор, принимающий делегат типа `Action<T>`, который помещается в оболочку:

```
var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
await Foo (progress);
```

(В классе `Progress<T>` также определено событие `ProgressChanged`, на которое можно подписаться вместо передачи делегата `Action` конструктору (или в дополнение к ней).) После создания экземпляра `Progress<int>` захватывается контекст синхронизации, если он существует. Когда метод `Foo` затем обращается к `Report`, делегат вызывается через упомянутый контекст.

Асинхронные методы могут реализовать более сложное сообщение о ходе работ путем замены `int` специальным типом, открывающим доступ к набору свойств.

Если вы знакомы с инфраструктурой Reactive Framework, то заметите, что интерфейс `IProgress<T>` вместе с типом задачи, возвращаемым асинхронной функцией, предоставляют набор средств, который подобен

такому набору, предлагаемому интерфейсом `IObserver<T>`. Отличие в том, что тип задачи может открывать доступ к "финальному" возвращаемому значению *в дополнение* к значениям (других типов), выдаваемым интерфейсом `IProgress<T>`.

Значения, выдаваемые `IProgress<T>`, обычно являются "одноразовыми" (скажем, процент выполненной работы или количество загруженных байтов), тогда как значения, возвращаемые методом `MoveNext` интерфейса `IObserver<T>`, обычно содержат в себе сам результат и поэтому существует веская причина для его вызова.

Асинхронные методы в WinRT также поддерживают возможность сообщения о ходе работ, хотя применяемый протокол сложнее из-за (относительно) отсталой системы типов COM. Взамен приема объекта, реализующего `IProgress<T>`, асинхронные методы WinRT, которые сообщают о ходе работ, возвращают на месте `IAsyncAction` и `IAsyncOperation<TResult>` один из следующих интерфейсов:

```
IAsyncActionWithProgress<TProgress>
IAsyncOperationWithProgress<TResult, TProgress>
```

Интересно отметить, что оба интерфейса основаны на `IAsyncInfo` (но не на `IAsyncAction` и `IAsyncOperation<TResult>`).

Хорошая новость в том, что расширяющий метод `AsTask` также перегружен, чтобы принимать `IProgress<T>` для указанных выше интерфейсов, поэтому потребители .NET могут игнорировать интерфейсы COM и поступать так, как показано ниже:

```
var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
CancellationToken cancelToken = ...
var task = someWinRTobject.FooAsync().AsTask (cancelToken, progress);
```

## Асинхронный шаблон, основанный на задачах

В .NET Framework 4.5 и последующих версиях доступны сотни асинхронных методов, возвращающих задачи, к которым можно применять `await` (они относятся главным образом к вводу-выводу). Большинство таких методов (по крайней мере, частично) следуют шаблону, который называется *асинхронным шаблоном, основанным на задачах* (Task-based Asynchronous Pattern – TAP), и представляет собой практичную формализацию всего того, что было описано до настоящего момента.

Метод TAP обладает следующими характеристиками:

- возвращает "горячий" (выполняющийся) экземпляр `Task` или `Task<TResult>`;
- имеет суффикс `Async` (за исключением специальных случаев, таких как комбинаторы задач);
- перегружен для приема признака отмены и/или `IProgress<T>`, если он поддерживает отмену и/или сообщение о ходе работ;
- быстро возвращает управление вызывающему коду (имеет только небольшую начальную синхронную фазу);
- не связывает поток, если является интенсивным в плане ввода-вывода.

Как видите, методы TAP легко писать с использованием асинхронных функций C#.

## Комбинаторы задач

Важным последствием наличия согласованного протокола для асинхронных функций (в соответствии с которым они возвращают объекты задач) является возможность применения и написания *комбинаторов задач* – функций, которые удобно объединяют задачи, не принимая во внимание то, что конкретно делает та или иная задача.

Среда CLR включает два комбинатора задач: `Task.WhenAny` и `Task.WhenAll`. При их описании мы будем предполагать, что определены следующие методы:

```
async Task<int> Delay1() { await Task.Delay (1000); return 1; }
async Task<int> Delay2() { await Task.Delay (2000); return 2; }
async Task<int> Delay3() { await Task.Delay (3000); return 3; }
```

### WhenAny

Метод `Task.WhenAny` возвращает объект задачи, которая завершается при завершении любой задачи из набора. В следующем примере задача завершается через одну секунду:

```
Task<int> winningTask = await Task.WhenAny (Delay1(), Delay2(), Delay3());
Console.WriteLine ("Done");
Console.WriteLine (winningTask.Result);  // 1
```

Поскольку метод `Task.WhenAny` сам возвращает объект задачи, мы применяем к его вызову `await`, что дает в итоге задачу, завершающуюся первой. Приведенный пример является полностью неблокирующим – включая последнюю строку, где производится доступ к свойству `Result` (т.к. задача `winningTask` уже будет завершена). Несмотря на это, обычно лучше применять `await` и к `winningTask`:

```
Console.WriteLine (await winningTask);  // 1
```

потому что тогда любое исключение генерируется повторно без помещения в оболочку `AggregateException`. На самом деле оба `await` могут находиться в одном операторе:

```
int answer = await await Task.WhenAny (Delay1(), Delay2(), Delay3());
```

Если какая-то из задач кроме завершившейся первой впоследствии откажет, то исключение станет необнаруженным, если только для объекта задачи не будет организовано ожидание посредством `await` (или не будет произведен доступ к его свойству `Exception`).

Метод `WhenAny` удобен для применения тайм-аутов или отмены к операциям, которые иначе подобное не поддерживают:

```
Task<string> task = SomeAsyncFunc();
Task winner = await (Task.WhenAny (task, Task.Delay(5000)));
if (winner != task) throw new TimeoutException();
string result = await task;    // Извлечь результат или повторно
                               // сгенерировать исключение
```

Обратите внимание, что поскольку в данном случае метод `WhenAny` вызывается с задачами разных типов, выигравшая задача возвращается как простой объект типа `Task` (а не `Task<string>`).

### WhenAll

Метод `Task.WhenAll` возвращает объект задачи, которая завершается, когда завершены *все* переданные ему задачи. В следующем примере задача завершается через три секунды (и демонстрируется шаблон *ветвления/присоединения* (fork/join)):

```
await Task.WhenAll (Delay1(), Delay2(), Delay3());
```

Похожий результат можно было бы получить без использования WhenAll, организовав ожидание task1, task2 и task3 по очереди:

```
Task task1 = Delay1(), task2 = Delay2(), task3 = Delay3();
await task1; await task2; await task3;
```

Отличие такого подхода (помимо меньшей эффективности из-за требования трех ожиданий вместо одного) связано с тем, что в случае отказа task1 мы никогда не перейдем к ожиданию задач task2/task3, и любые их исключения будут необнаруженными. На самом деле именно по такой причине поведение необнаруженных исключений, связанных с задачами, в версии CLR 4.5 стало менее строгим. Если вопреки наличию блока обработки исключений, охватывающего весь предыдущий код, исключение из задачи task2 или task3 могло бы привести к аварийному отказу приложения позже во время сборки мусора, тогда это определенно сбивало бы с толку.

Напротив, метод Task.WhenAll не завершается до тех пор, пока не будут завершены все задачи — даже когда возникает отказ. При появлении нескольких отказов их исключения объединяются в экземпляр AggregateException задачи (именно здесь класс AggregateException становится действительно полезным, потому что вы должны быть заинтересованы в получении всех исключений). Тем не менее, ожидание комбинированной задачи обеспечивает генерацию только первого исключения, так что для просмотра всех исключений понадобится поступить следующим образом:

```
Task task1 = Task.Run (() => { throw null; } );
Task task2 = Task.Run (() => { throw null; } );
Task all = Task.WhenAll (task1, task2);
try { await all; }
catch
{
  Console.WriteLine (all.Exception.InnerExceptions.Count);  // 2
}
```

Вызов WhenAll с задачами типа Task<TResult> возвращает Task<TResult[]>, предоставляя объединенные результаты всех задач. При ожидании все сводится к TResult[]:

```
Task<int> task1 = Task.Run (() => 1);
Task<int> task2 = Task.Run (() => 2);
int[] results = await Task.WhenAll (task1, task2);  // { 1, 2 }
```

В качестве практического примера рассмотрим параллельную загрузку веб-страниц по нескольким URI с подсчетом их суммарной длины:

```
async Task<int> GetTotalSize (string[] uris)
{
  IEnumerable<Task<byte[]>> downloadTasks = uris.Select (uri =>
    new WebClient().DownloadDataTaskAsync (uri));
  byte[][] contents = await Task.WhenAll (downloadTasks);
  return contents.Sum (c => c.Length);
}
```

Однако здесь присутствует некоторая неэффективность, связанная с тем, что во время загрузки мы излишне удерживаем байтовый массив до тех пор, пока не будет завершена каждая задача. Было бы более эффективно сразу же после загрузки сворачивать байтовые массивы в их длины. Для этого очень удобно применять асинхронные лямбда-выражения, потому что нам необходимо передавать выражение await в операцию запроса Select из LINQ:

```
async Task<int> GetTotalSize (string[] uris)
{
  IEnumerable<Task<int>> downloadTasks = uris.Select (async uri =>
    (await new WebClient().DownloadDataTaskAsync (uri)).Length);
  int[] contentLengths = await Task.WhenAll (downloadTasks);
  return contentLengths.Sum();
}
```

## Специальные комбинаторы

Временами удобно создавать собственные комбинаторы задач. Простейший "комбинатор" принимает одиночную задачу вроде приведенной ниже, что позволяет организовать ожидание любой задачи с использованием тайм-аута:

```
async static Task<TResult> WithTimeout<TResult> (this Task<TResult> task,
                                                 TimeSpan timeout)
{
  Task winner = await (Task.WhenAny (task, Task.Delay (timeout)));
  if (winner != task) throw new TimeoutException();
  return await task;  // Извлечь результат или повторно сгенерировать исключение
}
```

Следующий комбинатор позволяет "отменить" задачу посредством `CancellationToken`:

```
static Task<TResult> WithCancellation<TResult> (this Task<TResult> task,
                                                CancellationToken cancelToken)
{
  var tcs = new TaskCompletionSource<TResult>();
  var reg = cancelToken.Register (() => tcs.TrySetCanceled ());
  task.ContinueWith (ant =>
  {
    reg.Dispose();
    if (ant.IsCanceled)
      tcs.TrySetCanceled();
    else if (ant.IsFaulted)
      tcs.TrySetException (ant.Exception.InnerException);
    else
      tcs.TrySetResult (ant.Result);
  });
  return tcs.Task;
}
```

Комбинаторы задач могут оказаться сложными в написании, иногда требуя применения сигнализирующих конструкций, которые будут раскрыты в главе 22. На самом деле это хорошо, т.к. способствует вынесению сложности, связанной с параллелизмом, за пределы бизнес-логики и ее помещению в многократно используемые методы, которые могут быть протестированы в изоляции.

Следующий комбинатор работает подобно `WhenAll` за исключением того, что если любая из задач отказывает, то результирующая задача откажет незамедлительно:

```
async Task<TResult[]> WhenAllOrError<TResult>
    (params Task<TResult>[] tasks)
{
  var killJoy = new TaskCompletionSource<TResult[]>();
  foreach (var task in tasks)
    task.ContinueWith (ant =>
    {
      if (ant.IsCanceled)
        killJoy.TrySetCanceled();
```

```
      else if (ant.IsFaulted)
        killJoy.TrySetException (ant.Exception.InnerException);
    });
  return await await Task.WhenAny (killJoy.Task, Task.WhenAll (tasks));
}
```

Мы начинаем с создания экземпляра `TaskCompletionSource`, единственной работой которого является завершение всего в случае, если какая-то задача отказывает. Таким образом, мы никогда не вызываем его метод `SetResult`, а только методы `TrySetCanceled` и `TrySetException`. В данном случае метод `ContinueWith` более удобен, чем `GetAwaiter().OnCompleted`, потому что мы не обращаемся к результатам задач и в этой точке не хотим возврата в поток пользовательского интерфейса.

# Устаревшие шаблоны

В .NET Framework задействованы и другие шаблоны асинхронности, которые применялись до появления задач и асинхронных функций. Теперь они редко востребованы, поскольку с выходом версии .NET Framework 4.5 асинхронность на основе задач стала доминирующим шаблоном.

## Модель асинхронного программирования

Самый старый шаблон назывался *моделью асинхронного программирования* (Asynchronous Programming Model – APM) и использовал пару методов, имена которых начинаются с `Begin` и `End`, а также интерфейс по имени `IAsyncResult`. В целях иллюстрации мы возьмем класс `Stream` из пространства имен `System.IO` и рассмотрим его метод `Read`. Вначале взглянем на синхронную версию:

```
public int Read (byte[] buffer, int offset, int size);
```

Вероятно, вы уже в состоянии предугадать, каким образом выглядит асинхронная версия на основе *задач*:

```
public Task<int> ReadAsync (byte[] buffer, int offset, int size);
```

Теперь давайте посмотрим на версию APM:

```
public IAsyncResult BeginRead (byte[] buffer, int offset, int size,
                               AsyncCallback callback, object state);
public int EndRead (IAsyncResult asyncResult);
```

Вызов метода `Begin*` инициирует операцию, возвращая объект `IAsyncResult`, который действует в качестве признака для асинхронной операции. Когда операция завершается (или отказывает), запускается делегат `AsyncCallback`:

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Компонент, поддерживающий этот делегат, затем вызывает метод `End*`, который предоставляет возвращаемое значение операции, а также повторно генерирует исключение, если операция потерпела неудачу.

Шаблон APM не только неудобен в применении, но также неожиданно сложен в плане корректной реализации. Проще всего иметь дело с методами APM, вызывая метод адаптера `Task.Factory.FromAsync`, который преобразует пару методов APM в объект `Task`. Внутренне он использует `TaskCompletionSource`, чтобы предоставить объект задачи, которой отправляется сигнал, когда операция APM завершается или отказывает.

Метод FromAsync требует передачи следующих параметров:

- делегат, указывающий метод Begin*XXX*;
- делегат, указывающий метод End*XXX*;
- дополнительные аргументы, которые будут передаваться данным методам.

Метод FromAsync перегружен для приема типов делегатов и аргументов, которые соответствуют практически всем сигнатурам асинхронных методов, определенным в .NET Framework. Например, исходя из предположения, что stream имеет тип Stream, а buffer – тип byte[], мы можем записать так:

```
Task<int> readChunk = Task<int>.Factory.FromAsync (
  stream.BeginRead, stream.EndRead, buffer, 0, 1000, null);
```

### Асинхронные делегаты

Среда CLR по-прежнему поддерживает *асинхронные делегаты* – средство, которое позволяет вызывать любой делегат асинхронным образом с применением методов BeginInvoke/EndInvoke в стиле APM:

```
Func<string> foo = () => { Thread.Sleep(1000); return "foo"; };
foo.BeginInvoke (asyncResult =>
  Console.WriteLine (foo.EndInvoke (asyncResult)), null);
```

Асинхронные делегаты привносят неожиданно высокие накладные расходы и совершенно избыточны из-за наличия задач:

```
Func<string> foo = () => { Thread.Sleep(1000); return "foo"; };
Task.Run (foo).ContinueWith (ant => Console.WriteLine (ant.Result));
```

## Асинхронный шаблон на основе событий

*Асинхронный шаблон на основе событий* (Event-based Asynchronous Pattern – EAP) был введен в версии .NET Framework 2.0 для обеспечения более простой альтернативы шаблону APM, особенно в сценариях с пользовательским интерфейсом. Тем не менее, он был реализован лишь в небольшом количестве типов, наиболее примечательным из которых является WebClient в пространстве имен System.Net. Следует отметить, что EAP – это просто шаблон; никаких специальных типов для его поддержки не предусмотрено. По существу шаблон выглядит так: класс предлагает семейство членов, которые внутренне управляют параллелизмом, примерно как в показанном далее коде.

```
// Это члены класса WebClient:
public byte[] DownloadData (Uri address);
// Синхронная версия
public void DownloadDataAsync (Uri address);
public void DownloadDataAsync (Uri address, object userToken);
public event DownloadDataCompletedEventHandler DownloadDataCompleted;
public void CancelAsync (object userState);   // Отменяет операцию
public bool IsBusy { get; }   // Указывает, выполняется ли операция
```

Методы *Async инициируют выполнение операции асинхронным образом. Когда операция завершается, генерируется событие *Completed (с автоматической отправкой захваченному контексту синхронизации, если он имеется). Такое событие передает объект аргументов события, содержащий перечисленные ниже элементы:

- флаг, который указывает, была ли операция отменена (за счет вызова потребителем метода CancelAsync);

- объект Error, указывающий исключение, которое было сгенерировано (если было);
- объект userToken, если он предоставлялся при вызове метода *Async.

Типы EAP могут также определять событие сообщения о ходе работ, которое инициируется всякий раз, когда состояние хода работ изменяется (и также отправляется в контекст синхронизации):

```
public event DownloadProgressChangedEventHandler DownloadProgressChanged;
```

Реализация шаблона EAP требует написания большого объема стереотипного кода, делая этот шаблон неудобным с композиционной точки зрения.

## BackgroundWorker

Универсальной реализацией шаблона EAP является класс BackgroundWorker из пространства имен System.ComponentModel. Он позволяет обогащенным клиентским приложениям запускать рабочий поток и сообщать о проценте выполненной работы без необходимости в явном захвате контекста синхронизации. Например:

```
var worker = new BackgroundWorker { WorkerSupportsCancellation = true };
worker.DoWork += (sender, args) =>
{ // Выполняется в рабочем потоке
  if (args.Cancel) return;
  Thread.Sleep(1000);
  args.Result = 123;
};

worker.RunWorkerCompleted += (sender, args) =>
{ // Выполняется в потоке пользовательского интерфейса
  // Здесь можно безопасно обновлять элементы управления
  // пользовательского интерфейса...
  if (args.Cancelled)
    Console.WriteLine ("Cancelled");
  else if (args.Error != null)
    Console.WriteLine ("Error: " + args.Error.Message);
  else
    Console.WriteLine ("Result is: " + args.Result);
};

worker.RunWorkerAsync();      // Захватывает контекст синхронизации
                              // и запускает операцию
```

Метод RunWorkerAsync запускает операцию, инициируя событие DoWork в рабочем потоке из пула. Он также захватывает контекст синхронизации, и когда операция завершается (или отказывает), через данный контекст генерируется событие RunWorkerCompleted (подобно признаку продолжения).

Класс BackgroundWorker порождает крупномодульный параллелизм, при котором событие DoWork инициируется полностью в рабочем потоке. Если в этом обработчике событий нужно обновлять элементы управления пользовательского интерфейса (помимо отправки сообщения о проценте выполненных работ), тогда придется использовать Dispatcher.BeginInvoke или похожий метод.

Класс BackgroundWorker более подробно описан в статье по адресу www.albahari.com/threading.

# 15

# Потоки данных и ввод-вывод

В настоящей главе описаны фундаментальные типы, предназначенные для ввода и вывода в .NET, с акцентированием внимания на следующих темах:

- потоковая архитектура .NET и предоставление ею согласованного программного интерфейса для чтения и записи с применением разнообразных типов ввода-вывода;
- классы для манипулирования файлами и каталогами на диске;
- специализированные потоки для сжатия, именованные каналы и размещенные в памяти файлы.

Глава сконцентрирована на типах из пространства имен `System.IO`, где находится функциональность ввода-вывода самого низкого уровня. Инфраструктура .NET Framework также предлагает функциональность ввода-вывода более высокого уровня в форме подключений и команд SQL, LINQ to SQL и LINQ to XML, Windows Communication Foundation, Web Services и Remoting.

## Потоковая архитектура

Потоковая архитектура .NET основана на трех концепциях: опорные хранилища, декораторы и адаптеры (рис. 15.1).

*Опорное хранилище* представляет собой конечную точку, которая делает ввод и вывод полезными, такая как файл или сетевое подключение. Точнее, это один или оба следующих компонента:

- источник, с которого могут последовательно читаться байты;
- приемник, куда байты могут последовательно записываться.

Тем не менее, опорное хранилище не может использоваться, если программисту не открыт к нему доступ. Стандартным классом .NET, который предназначен для такой цели, является `Stream`; он предоставляет стандартный набор методов, позволяющих выполнять чтение, запись и позиционирование.

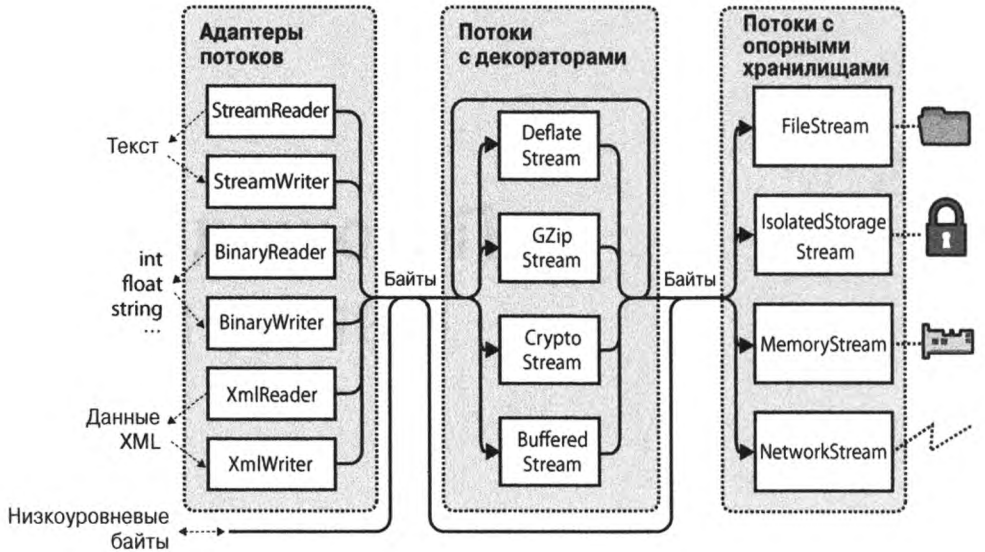



***Рис. 15.1.** Потоковая архитектура*

В отличие от массива, где все опорные данные существуют в памяти одновременно, поток имеет дело с данными последовательно – либо по одному байту за раз, либо в блоках управляемого размера. Следовательно, поток может потреблять мало памяти независимо от размера его опорного хранилища.

Потоки делятся на две категории.

- *Потоки с опорными хранилищами.* Потоки, которые жестко привязаны к определенному типу опорного хранилища, такие как `FileStream` или `NetworkStream`.
- *Потоки с декораторами.* Потоки, которые наполняют другие потоки, каким-то образом трансформируя данные, например, `DeflateStream` или `CryptoStream`.

Потоки с декораторами обладают перечисленными ниже архитектурными преимуществами:

- они освобождают потоки с опорными хранилищами от необходимости самостоятельной реализации таких возможностей, как сжатие и шифрование;
- потоки не страдают от изменения интерфейса, когда они декорированы;
- декораторы можно подключать во время выполнения;
- декораторы можно соединять в цепочки (скажем, декоратор сжатия можно соединить с декоратором шифрования).

Потоки с опорными хранилищами и потоки с декораторами имеют дело исключительно с байтами. Хотя это гибко и эффективно, приложения часто работают на более высоких уровнях, таких как текст или XML. *Адаптеры* преодолевают такой разрыв, помещая поток в оболочку класса со специализированными методами, которые типизированы для конкретного формата. Например, средство чтения текста открывает доступ к методу `ReadLine`, а средство записи XML – к методу `WriteAttributes`.

Адаптер помещает поток внутрь оболочки в точности как декоратор. Однако в отличие от декоратора адаптер *сам по себе* не является потоком; он обычно полностью скрывает байт-ориентированные методы.

Подведем итоги: потоки с опорными хранилищами предоставляют низкоуровневые данные; потоки с декораторами обеспечивают прозрачные двоичные трансформации вроде шифрования; адаптеры предлагают типизированные методы для работы с типами более высокого уровня, такими как строки и XML. Связи между ними проиллюстрированы на рис. 15.1. Чтобы сформировать цепочку, необходимо просто передать один объект конструктору другого класса.

# Использование потоков

Абстрактный класс `Stream` является базовым для всех потоков. В нем определены методы и свойства для трех фундаментальных операций: *чтение*, *запись* и *поиск*, а также для выполнения административных задач, подобных закрытию, сбрасыванию и конфигурированию тайм-аутов (табл. 15.1).

**Таблица 15.1. Члены класса `Stream`**

| Категория | Члены |
|---|---|
| Чтение | `public abstract bool CanRead { get; }`<br>`public abstract int Read (byte[] buffer, int offset, int count)`<br>`public virtual int ReadByte();` |
| Запись | `public abstract bool CanWrite { get; }`<br>`public abstract void Write (byte[] buffer, int offset, int count);`<br>`public virtual void WriteByte (byte value);` |
| Поиск | `public abstract bool CanSeek { get; }`<br>`public abstract long Position { get; set; }`<br>`public abstract void SetLength (long value);`<br>`public abstract long Length { get; }`<br>`public abstract long Seek (long offset, SeekOrigin origin);` |
| Закрытие/ сбрасывание | `public virtual void Close();`<br>`public void Dispose();`<br>`public abstract void Flush();` |
| Тайм-ауты | `public virtual bool CanTimeout { get; }`<br>`public virtual int ReadTimeout { get; set; }`<br>`public virtual int WriteTimeout { get; set; }` |
| Другие | `public static readonly Stream Null; // Поток null`<br>`public static Stream Synchronized (Stream stream);` |

Начиная с версии .NET Framework 4.5, доступны также асинхронные версии методов `Read` и `Write`, которые возвращают объекты `Task` и дополнительно принимают признак отмены.

В следующем примере демонстрируется применение файлового потока для чтения, записи и позиционирования:

```
using System;
using System.IO;

class Program
{
  static void Main()
  {
    // Создать файл по имени test.txt в текущем каталоге:
    using (Stream s = new FileStream ("test.txt", FileMode.Create))
    {
      Console.WriteLine (s.CanRead);    // True
      Console.WriteLine (s.CanWrite);   // True
      Console.WriteLine (s.CanSeek);    // True
      s.WriteByte (101);
      s.WriteByte (102);
      byte[] block = { 1, 2, 3, 4, 5 };
      s.Write (block, 0, block.Length); // Записать блок из 5 байтов
      Console.WriteLine (s.Length);     // 7
      Console.WriteLine (s.Position);   // 7
      s.Position = 0;                   // Переместиться обратно в начало
      Console.WriteLine (s.ReadByte()); // 101
      Console.WriteLine (s.ReadByte()); // 102

      // Читать из потока в массив block:
      Console.WriteLine (s.Read (block, 0, block.Length));  // 5

      // Предполагая, что последний вызов Read возвратил 5,
      // мы находимся в конце файла, и Read теперь возвратит 0:
      Console.WriteLine (s.Read (block, 0, block.Length));  // 0
    }
  }
}
```

Асинхронное чтение или запись предусматривает просто вызов метода `ReadAsync/WriteAsync` вместо `Read/Write` и применение к выражению ключевого слова `await`. (К вызываемому методу потребуется также добавить ключевое слово `async`, как объяснялось в главе 14.)

```
async static void AsyncDemo()
{
  using (Stream s = new FileStream ("test.txt", FileMode.Create))
  {
    byte[] block = { 1, 2, 3, 4, 5 };
    await s.WriteAsync (block, 0, block.Length); // Выполнить запись асинхронно
    s.Position = 0;  // Переместиться обратно в начало

    // Читать из потока в массив block:
    Console.WriteLine (await s.ReadAsync (block, 0, block.Length));  // 5
  }
}
```

Асинхронные методы упрощают построение отзывчивых и масштабируемых приложений, которые работают с потенциально медленными потоками данных (особенно сетевыми потоками), не связывая поток управления.

Для краткости мы будем использовать синхронные методы почти во всех примерах настоящей главы; тем не менее, в большинстве сценариев, связанных с сетевым вводом-выводом, мы рекомендуем отдавать предпочтение асинхронным операциям `Read/Write`.

## Чтение и запись

Поток может поддерживать чтение, запись или то и другое. Если свойство `CanWrite` возвращает `false`, тогда поток предназначен только для чтения; если свойство `CanRead` возвращает `false`, то поток предназначен только для записи.

Метод `Read` получает блок данных из потока и помещает его в массив. Он возвращает количество полученных байтов, которое всегда либо меньше, либо равно значению аргумента `count`. Если оно меньше `count`, то это означает, что достигнут конец потока или поток выдает данные порциями меньшего размера (как часто случается с сетевыми потоками). В любом случае остаток байтов в массиве останется неизменным, сохраняя предыдущие значения.

При работе с методом `Read` можно определенно утверждать, что достигнут конец потока, только когда он возвращает 0. Таким образом, если есть поток из 1000 байтов, тогда следующий код может не прочитать их все в память:

```
// Предполагается, что s является потоком:
byte[] data = new byte [1000];
s.Read (data, 0, data.Length);
```

Метод `Read` мог бы прочитать от 1 до 1000 байтов, оставив остаток потока непрочитанным.

Вот корректный способ чтения потока из 1000 байтов:

```
byte[] data = new byte [1000];

// Переменная bytesRead в итоге получит значение 1000,
// если только сам поток не имеет меньшую длину:

int bytesRead = 0;
int chunkSize = 1;
while (bytesRead < data.Length && chunkSize > 0)
  bytesRead +=
    chunkSize = s.Read (data, bytesRead, data.Length - bytesRead);
```

К счастью, тип `BinaryReader` предлагает более простой способ для достижения того же результата:

```
byte[] data = new BinaryReader (s).ReadBytes (1000);
```

Если поток имеет длину меньше 1000 байтов, то возвращенный байтовый массив отражает действительный размер потока. Если поток поддерживает поиск, тогда можно прочитать все его содержимое, заменив `1000` выражением `(int)s.Length`.

Мы более подробно опишем тип `BinaryReader` в разделе "Адаптеры потоков" далее в главе.

Метод `ReadByte` проще: он читает одиночный байт, возвращая –1 для указания на конец массива. На самом деле `ReadByte` возвращает значение типа `int`, а не `byte`, т.к. тип `byte` значение –1 не поддерживает.

Методы `Write` и `WriteByte` отправляют данные в поток. Если они не могут отправить указанные байты, то генерируется исключение.

В методах `Read` и `Write` аргумент `offset` ссылается на индекс в массиве `buffer`, с которого начинается чтение или запись, а не на позицию внутри потока.

## Поиск

Поток поддерживает возможность позиционирования, если свойство `CanSeek` возвращает `true`. Для потока с возможностью позиционирования (такого как файловый поток) можно запрашивать или модифицировать его свойство `Length` (вызывая метод `SetLength`) и в любой момент изменять свойство `Position`, отражающее позицию, в которой производится чтение или запись. Свойство `Position` принимает значения относительно начала потока; однако метод `Seek` позволяет перемещаться относительно текущей позиции или относительно конца потока.

Изменение свойства `Position` экземпляра `FileStream` обычно занимает несколько микросекунд. Если вам нужно делать это миллионы раз в цикле, тогда класс `MemoryMappedFile` может оказаться более удачным выбором, чем `FileStream` (как показано в разделе "Размещенные в памяти файлы" далее в главе).

Единственный способ определения длины потока, не поддерживающего возможность позиционирования (вроде потока с шифрованием), заключается в его чтении до самого конца. Более того, если требуется повторно прочитать предшествующую область, то поток придется закрыть и начать работу с новым потоком.

## Закрытие и сбрасывание

После применения потоки должны быть освобождены, чтобы освободить лежащие в их основе ресурсы, подобные файловым и сокетным дескрипторам. Самый простой способ предусматривает создание экземпляров потоков внутри блоков `using`. В общем случае потоки поддерживают следующую стандартную семантику освобождения:

- методы `Dispose` и `Close` функционируют идентично;
- многократное освобождение или закрытие потока не вызывает ошибки.

Закрытие потока с декоратором приводит к закрытию и декоратора, и его потока с опорным хранилищем. В случае цепочки декораторов закрытие самого внешнего декоратора (в голове цепочки) закрывает всю цепочку.

Некоторые потоки внутренне буферизуют данные, поступающие в и из опорного хранилища, чтобы снизить количество двухсторонних обменов и тем самым улучшить производительность (хорошим примером служат файловые потоки). Это значит, что данные, записываемые в поток, могут не сразу попасть в опорное хранилище; возможна задержка до тех пор, пока буфер не заполнится. Метод `Flush` обеспечивает принудительную запись любых буферизированных данных. Метод `Flush` вызывается автоматически при закрытии потока, потому поступать так, как показано ниже, никогда не придется:

```
s.Flush(); s.Close();
```

## Тайм-ауты

Поток поддерживает тайм-ауты чтения и записи, если свойство `CanTimeout` возвращает `true`. Сетевые потоки поддерживают тайм-ауты, а файловые потоки и потоки в памяти — нет. Для потоков, поддерживающих тайм-ауты, свойства `ReadTimeout`

и `WriteTimeout` задают желаемый тайм-аут в миллисекундах, причем 0 означает отсутствие тайм-аута. Методы `Read` и `Write` указывают на то, что тайм-аут произошел, генерацией исключения.

## Безопасность в отношении потоков управления

Как правило, потоки данных не являются безопасными в отношении потоков управления, т.е. два потока управления не могут параллельно выполнять чтение или запись в один и тот же поток данных, не создавая возможность для ошибки. Класс `Stream` предлагает простой обходной путь через статический метод `Synchronized`, который принимает поток данных любого типа и возвращает оболочку, безопасную к потокам управления. Такая оболочка работает за счет получения монопольной блокировки на каждой операции чтения, записи или позиционирования, гарантируя, что в любой момент времени заданную операцию может выполнять только один поток управления. На практике это позволяет множеству потоков управления одновременно дописывать данные в один и тот же поток данных – другие разновидности действий (подобные параллельному чтению) требуют дополнительной блокировки, обеспечивающей доступ каждого потока управления к желаемой части потока данных. Вопросы безопасности в отношении потоков управления подробно обсуждаются в главе 22.

## Потоки с опорными хранилищами

На рис. 15.2 показаны основные потоки с опорными хранилищами, предлагаемые .NET Framework. "Поток `null`" также доступен через статическое поле `Null` класса `Stream`. Потоки `null` могут быть удобны при написании модульных тестов.

В последующих разделах мы опишем классы `FileStream` и `MemoryStream`, а в финальном разделе настоящей главы – класс `IsolatedStorageStream`. Класс `NetworkStream` будет раскрыт в главе 16.

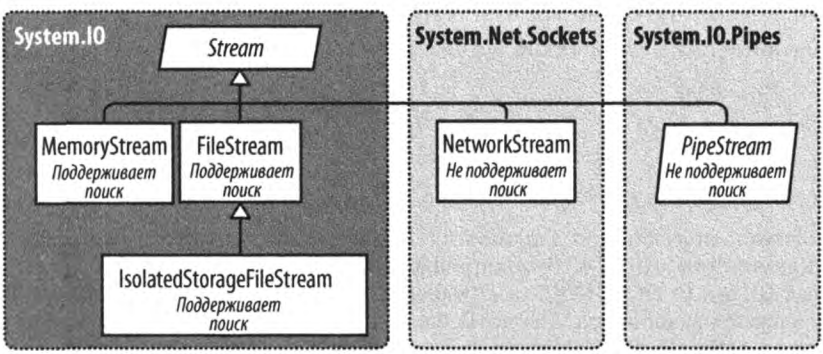



***Рис. 15.2.*** *Основные потоки с опорными хранилищами*

### `FileStream`

Ранее в разделе мы демонстрировали базовое использование класса `FileStream` для чтения и записи байтов данных. Теперь мы рассмотрим специальные возможности этого класса.

В случае применения UWP файловый ввод-вывод лучше выполнять с помощью типов Windows Runtime из пространства имен `Windows.Storage` (см. раздел "Файловый ввод-вывод в UWP" далее в главе).

### Конструирование экземпляра FileStream

Простейший способ создания экземпляра FileStream предполагает использование следующих статических фасадных методов класса File:

```
FileStream fs1 = File.OpenRead ("readme.bin");          // Только для чтения
FileStream fs2 = File.OpenWrite (@"c:\temp\writeme.tmp"); // Только для записи
FileStream fs3 = File.Create (@"c:\temp\writeme.tmp"); //Для чтения и записи
```

Поведение методов OpenWrite и Create отличается в ситуации, когда файл уже существует. Метод Create усекает любое имеющееся содержимое, а метод OpenWrite оставляет содержимое незатронутым, устанавливая позицию потока в ноль. Если будет записано меньше байтов, чем ранее существовало в файле, то метод OpenWrite оставит смесь старого и нового содержимого.

Создавать экземпляры FileStream можно также напрямую. Конструкторы класса FileStream предоставляют доступ ко всем средствам, позволяя указывать имя файла или низкоуровневый файловый дескриптор, режимы создания и доступа к файлу, а также опции для совместного использования, буферизации и безопасности. Приведенный ниже оператор открывает существующий файл для чтения/записи, не перезаписывая его:

```
var fs = new FileStream ("readwrite.tmp", FileMode.Open);  // Чтение/запись
```

Вскоре мы более подробно рассмотрим перечисление FileMode.

---

### Сокращенные методы класса File

Следующие статические методы читают целый файл в память за один шаг:

- File.ReadAllText (возвращает строку);
- File.ReadAllLines (возвращает массив строк);
- File.ReadAllBytes (возвращает байтовый массив).

Приведенные ниже статические методы записывают целый файл за один шаг:

- File.WriteAllText;
- File.WriteAllLines;
- File.WriteAllBytes;
- File.AppendAllText (удобен для добавления данных в журнальный файл).

Есть также статический метод по имени File.ReadLines: он похож на ReadAllLines за исключением того, что возвращает лениво оцениваемое перечисление IEnumerable<string>. Он более эффективен, т.к. не производит загрузку всего файла в память за один раз. Для потребления результатов идеально подходит LINQ; скажем, следующий код подсчитывает количество строк с длиной, превышающей 80 символов:

```
int longLines = File.ReadLines ("filePath")
                    .Count (l => l.Length > 80);
```

---

### Указание имени файла

Имя файла может быть либо абсолютным (например, c:\temp\test.txt), либо относительным к текущему каталогу (например, test.txt или temp\test.txt). Получить доступ или изменить текущий каталог можно через статическое свойство Environment.CurrentDirectory.

Когда программа запускается, текущий каталог может совпадать или не совпадать с каталогом, где находится исполняемый файл программы. По этой причине при поиске дополнительных файлов времени выполнения, упакованных вместе с исполняемым файлом, никогда не следует полагаться на текущий каталог.

Свойство `AppDomain.CurrentDomain.BaseDirectory` возвращает *базовый каталог приложения*, которым в нормальных случаях является папка, содержащая исполняемый файл программы. Чтобы указать имя файла относительно базового каталога приложения, можно вызвать метод `Path.Combine`:

```
string baseFolder = AppDomain.CurrentDomain.BaseDirectory;
string logoPath = Path.Combine (baseFolder, "logo.jpg");
Console.WriteLine (File.Exists (logoPath));
```

Можно выполнять чтение и запись по сети через путь UNC, такой как `\\JoesPC\PicShare\pic.jpg` или `\\10.1.1.2\PicShare\pic.jpg`.

### Указание режима файла

Все конструкторы класса `FileStream`, которые принимают имя файла, также требуют указания режима файла – аргумента типа перечисления `FileMode`. На рис. 15.3 показано, как выбрать значение `FileMode`, и варианты дают результаты сродни вызову статического метода класса `File`.

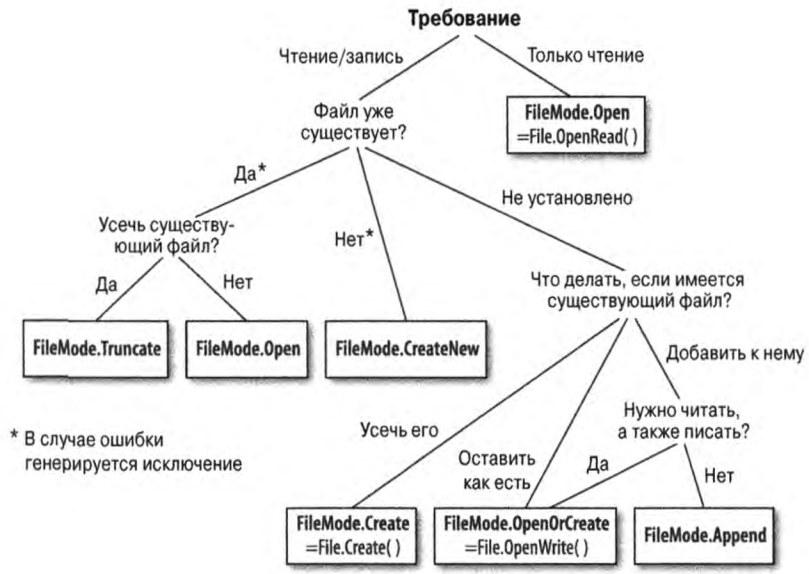



***Рис. 15.3.** Выбор значения `FileMode`*

Метод `File.Create` и значение `FileMode.Create` приведут к генерации исключения, если используются для скрытых файлов. Чтобы перезаписать скрытый файл, потребуется удалить его и затем создать повторно:

```
if (File.Exists ("hidden.txt")) File.Delete ("hidden.txt");
```

Конструирование экземпляра FileStream с указанием имени файла и режима FileMode дает (с одним исключением) поток с возможностью чтения/записи. Можно запросить понижение уровня доступа, если также предоставить аргумент FileAccess:

```
[Flags]
public enum FileAccess { Read = 1, Write = 2, ReadWrite = 3 }
```

Следующий вызов возвращает поток, предназначенный только для чтения, и эквивалентен вызову метода File.OpenRead:

```
using (var fs = new FileStream ("x.bin", FileMode.Open, FileAccess.Read))
  ...
```

Значение FileMode.Append считается особым: в таком режиме будет получен поток, предназначенный *только для записи*. Чтобы можно было добавлять, располагая поддержкой чтения-записи, вместо FileMode.Append придется указать FileMode.Open или FileMode.OpenOrCreate и перейти в конец потока:

```
using (var fs = new FileStream ("myFile.bin", FileMode.Open))
{
  fs.Seek (0, SeekOrigin.End);
  ...
```

### Расширенные возможности FileStream

Ниже описаны другие необязательные аргументы, которые можно задавать при конструировании экземпляра FileStream.

- Значение перечисления FileShare, которое описывает, какой уровень доступа должен быть выдан другим процессам, чтобы они могли просматривать файл до того, как вы завершите с ним работу (None, Read (по умолчанию), ReadWrite или Write).
- Размер внутреннего буфера в байтах (в настоящее время стандартным является размер, составляющий 4 Кбайт).
- Флаг, который указывает, следует ли возложить асинхронный вывод на операционную систему (ОС).
- Объект FileSecurity, описывающий права доступа на уровне пользователей и ролей для назначения новому файлу.
- Значение перечисления флагов FileOptions для запроса шифрования ОС (Encrypted), автоматического удаления при закрытии временных файлов (DeleteOnClose) и подсказки для оптимизации (RandomAccess и SequentialScan). Имеется также флаг WriteThrough, который запрашивает у ОС отключение кеширования при записи; он предназначен для транзакционных файлов или журналов.

Открытие файла со значением FileShare.ReadWrite позволяет другим процессам или пользователям одновременно читать и записывать в один и тот же файл. Во избежание хаоса потребуется блокировать определенные области файла перед чтением или записью с помощью следующих методов:

```
// Определены в классе FileStream:
public virtual void Lock   (long position, long length);
public virtual void Unlock (long position, long length);
```

Метод Lock генерирует исключение, если часть или вся запрошенная область файла уже заблокирована.

## MemoryStream

В качестве опорного хранилища класс MemoryStream использует массив. Отчасти это противоречит замыслу самого потока, поскольку опорное хранилище должно располагаться в памяти целиком. Класс MemoryStream все еще полезен, когда необходим произвольный доступ в поток данных, не поддерживающий позиционирование. Если известно, что исходный поток будет иметь поддающийся управлению размер, то вот как его можно скопировать в MemoryStream:

```
var ms = new MemoryStream();
sourceStream.CopyTo (ms);
```

Вызвав метод ToArray, поток MemoryStream можно преобразовать в байтовый массив. Метод GetBuffer делает ту же самую работу более эффективно, возвращая прямую ссылку на лежащий в основе массив хранилища; к сожалению, такой массив обычно превышает реальный размер потока.

Закрывать и сбрасывать MemoryStream не обязательно. После закрытия потока MemoryStream производить чтение и запись в него больше не удастся, но по-прежнему можно вызывать метод ToArray для получения лежащих в основе данных. Метод Flush в потоке MemoryStream вообще ничего не делает.

Дополнительные примеры использования MemoryStream можно найти в разделе "Потоки со сжатием" далее в главе и в разделе "Обзор криптографии" главы 21.

## PipeStream

Класс PipeStream появился в версии .NET Framework 3.5. Он предоставляет простой способ взаимодействия одного процесса с другим через протокол *каналов* Windows.

Различают два вида каналов.

- *Анонимный канал.* Делает возможным однонаправленное взаимодействие между родительским и дочерним процессом на одном и том же компьютере.
- *Именованный канал (более гибкий).* Делает возможным двунаправленное взаимодействие между произвольными процессами на одном и том же компьютере или на разных компьютерах по сети Windows.

Канал удобен для организации взаимодействия между процессами (interprocess communication – IPC) на одном компьютере: он не полагается на сетевой транспорт, что означает отсутствие накладных расходов, связанных с протоколами, и проблем с брандмауэрами.

Каналы основаны на потоках, так что один процесс ожидает получения последовательности байтов, в то время как другой процесс их отправляет. Альтернативной является взаимодействие процессов через блок разделяемой памяти – мы покажем, как это делать, в разделе "Размещенные в памяти файлы" далее в главе.

Тип PipeStream представляет собой абстрактный класс с четырьмя конкретными подтипами. Два из них применяются для анонимных каналов и еще два – для именованных каналов.

**Анонимные каналы**

`AnonymousPipeServerStream` и `AnonymousPipeClientStream`

**Именованные каналы**

`NamedPipeServerStream` и `NamedPipeClientStream`

Именованные каналы проще в использовании, так что рассмотрим их первыми.

Канал — это низкоуровневая конструкция, которая позволяет отправлять и получать байты (или *сообщения*, являющиеся группами байтов). API-интерфейсы WCF и Remoting предлагают высокоуровневые инфраструктуры обмена сообщениями с возможностью применения канала IPC для взаимодействия.

## Именованные потоки

В случае именованных потоков участники взаимодействуют через канал с таким же именем. Протокол определяет две отдельные роли: клиент и сервер. Взаимодействие между клиентом и сервером происходит следующим образом.

- Сервер создает экземпляр `NamedPipeServerStream` и вызывает метод `WaitForConnection`.
- Клиент создает экземпляр `NamedPipeClientStream` и вызывает метод `Connect` (необязательно указывая тайм-аут).

Затем для взаимодействия два участника производят чтение и запись в потоки.

В приведенном ниже примере демонстрируется сервер, который отправляет одиночный байт (100) и ожидает получения одиночного байта:

```
using (var s = new NamedPipeServerStream ("pipedream"))
{
  s.WaitForConnection();
  s.WriteByte (100);      // Отправить значение 100
  Console.WriteLine (s.ReadByte());
}
```

А вот соответствующий код клиента:

```
using (var s = new NamedPipeClientStream ("pipedream"))
{
  s.Connect();
  Console.WriteLine (s.ReadByte());
  s.WriteByte (200);      // Отправить обратно значение 200
}
```

Потоки именованных каналов по умолчанию являются двунаправленными, так что любой из участников может читать или записывать в свой поток. Это значит, что клиент и сервер должны следовать определенному протоколу для координирования своих действий, чтобы оба участника не начали одновременно отправлять или получать данные.

Также должно быть предусмотрено соглашение о длине каждой передачи. В данном смысле приведенный выше пример тривиален, поскольку в каждом направлении передается один байт. Для поддержки сообщений длиннее одного байта каналы предлагают режим передачи *сообщений*. Когда он включен, вызывающий метод `Read` участник может узнать о том, что сообщение завершено, проверив свойство `IsMessageComplete`.

В целях демонстрации начнем с написания вспомогательного метода, который читает целое сообщение из PipeStream с включенным режимом передачи сообщений — другими словами, до тех пор, пока свойство IsMessageComplete не станет равным true:

```
static byte[] ReadMessage (PipeStream s)
{
  MemoryStream ms = new MemoryStream();
  byte[] buffer = new byte [0x1000];  // Читать блоками по 4 Кбайт

  do    { ms.Write (buffer, 0, s.Read (buffer, 0, buffer.Length)); }
  while (!s.IsMessageComplete);

  return ms.ToArray();
}
```

(Чтобы сделать код асинхронным, замените s.Read конструкцией await s.ReadAsync.)

Просто ожидая возвращения методом Read значения 0, нельзя выяснить, завершил ли поток PipeStream чтение сообщения. Причина в том, что в отличие от большинства других типов потоков потоки каналов и сетевые потоки не имеют четко выраженного окончания. Взамен они временно "опустошаются" между передачами сообщений.

Теперь можно активизировать режим передачи сообщений. На стороне сервера это делается за счет указания PipeTransmissionMode.Message во время конструирования потока:

```
using (var s = new NamedPipeServerStream ("pipedream", PipeDirection.InOut,
                                           1, PipeTransmissionMode.Message))
{
  s.WaitForConnection();

  byte[] msg = Encoding.UTF8.GetBytes ("Hello");
  s.Write (msg, 0, msg.Length);

  Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));
}
```

На стороне клиента режим передачи сообщений включается установкой свойства ReadMode после вызова метода Connect:

```
using (var s = new NamedPipeClientStream ("pipedream"))
{
  s.Connect();
  s.ReadMode = PipeTransmissionMode.Message;

  Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));

  byte[] msg = Encoding.UTF8.GetBytes ("Hello right back!");
  s.Write (msg, 0, msg.Length);
}
```

### Анонимные каналы

Анонимный канал предоставляет однонаправленный поток взаимодействия между родительским и дочерним процессами. Вместо использования имени на уровне системы анонимные каналы настраиваются посредством закрытого дескриптора.

Как и именованные каналы, анонимные каналы имеют отдельные роли клиента и сервера. Однако система взаимодействия несколько отличается и происходит следующим образом.

1. Сервер создает экземпляр класса `AnonymousPipeServerStream`, фиксируя направление канала (`PipeDirection`) как `In` или `Out`.
2. Сервер вызывает метод `GetClientHandleAsString`, чтобы получить идентификатор для канала, который затем передается клиенту (обычно в качестве аргумента при запуске дочернего процесса).
3. Дочерний процесс создает экземпляр класса `AnonymousPipeClientStream`, указывая противоположное направление канала (`PipeDirection`).
4. Сервер освобождает локальный дескриптор, который был сгенерирован на шаге 2, вызывая метод `DisposeLocalCopyOfClientHandle`.
5. Родительский и дочерний процессы взаимодействуют, читая/записывая в поток.

Поскольку анонимные каналы являются однонаправленными, для двунаправленного взаимодействия сервер должен создать два канала. Приведенный далее код реализует сервер, который отправляет одиночный байт дочернему процессу, а затем получает от него также одиночный байт:

```
string clientExe = @"d:\PipeDemo\ClientDemo.exe";

HandleInheritability inherit = HandleInheritability.Inheritable;

using (var tx = new AnonymousPipeServerStream (PipeDirection.Out, inherit))
using (var rx = new AnonymousPipeServerStream (PipeDirection.In, inherit))
{
  string txID = tx.GetClientHandleAsString();
  string rxID = rx.GetClientHandleAsString();

  var startInfo = new ProcessStartInfo (clientExe, txID + " " + rxID);
  startInfo.UseShellExecute = false;     // Требуется для дочернего процесса
  Process p = Process.Start (startInfo);

  tx.DisposeLocalCopyOfClientHandle();  // Освободить неуправляемые
  rx.DisposeLocalCopyOfClientHandle();  //   ресурсы дескрипторов

  tx.WriteByte (100);
  Console.WriteLine ("Server received: " + rx.ReadByte());

  p.WaitForExit();
}
```

Ниже показан код соответствующего клиента, который должен быть скомпилирован в исполняемый файл `d:\PipeDemo\ClientDemo.exe`:

```
string rxID = args[0];  // Обратите внимание на смену
string txID = args[1];  //   ролей приема и передачи

using (var rx = new AnonymousPipeClientStream (PipeDirection.In, rxID))
using (var tx = new AnonymousPipeClientStream (PipeDirection.Out, txID))
{
  Console.WriteLine ("Client received: " + rx.ReadByte());
  tx.WriteByte (200);
}
```

Как и в случае именованных каналов, клиент и сервер должны координировать свои отправки и получения и согласовывать длину каждой передачи. К сожалению,

анонимные каналы не поддерживают режим передачи сообщений, а потому вам придется реализовать собственный протокол для согласования длины сообщений. Одним из решений может быть отправка в первых 4 байтах каждой передачи целочисленного значения, которое определяет длину сообщения, следующего за этими 4 байтами. Класс `BitConverter` предоставляет методы для преобразования между целочисленным типом и массивом из 4 байтов.

## BufferedStream

Класс `BufferedStream` декорирует, или помещает в оболочку, другой поток, добавляя возможность буферизации, и является одним из нескольких типов потоков с декораторами, которые определены в ядре .NET Framework; все типы показаны на рис. 15.4.

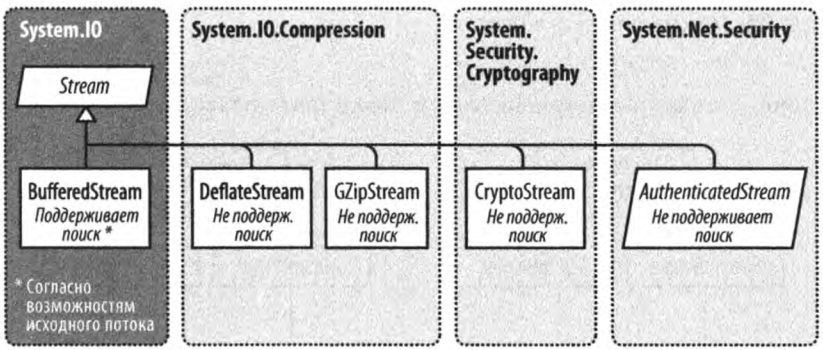



***Рис. 15.4.** Потоки с декораторами*

Буферизация улучшает производительность, сокращая количество двухсторонних обменов с опорным хранилищем. Ниже показано, как поместить поток `FileStream` в `BufferedStream` с буфером 20 Кбайт:

```
// Записать 100 000 байтов в файл:
File.WriteAllBytes ("myFile.bin", new byte [100000]);

using (FileStream fs = File.OpenRead ("myFile.bin"))
using (BufferedStream bs = new BufferedStream (fs, 20000)) // Буфер размером
                                                           // 20 Кбайт
{
  bs.ReadByte();
  Console.WriteLine (fs.Position);  // 20000
}
```

В приведенном примере благодаря буферизации с опережающим чтением внутренний поток перемещает 20 000 байтов после чтения только одного байта. Вызывать метод `ReadByte` можно было бы еще 19 999 раз, и лишь тогда снова произошло бы обращение к `FileStream`.

Связывание `BufferedStream` с `FileStream`, как в предыдущем примере, не особенно ценно, т.к. класс `FileStream` сам поддерживает встроенную буферизацию. Оно могло понадобиться единственно для расширения буфера уже сконструированного потока `FileStream`.

Закрытие `BufferedStream` автоматически закрывает лежащий в основе поток с опорным хранилищем.

# Адаптеры потоков

Класс `Stream` имеет дело только с байтами; для чтения и записи таких типов данных, как строки, целые числа или XML-элементы, потребуется подключить адаптер. Ниже описаны виды адаптеров, предлагаемые .NET Framework.

**Текстовые адаптеры (для строковых и символьных данных)**

```
TextReader, TextWriter
StreamReader, StreamWriter
StringReader, StringWriter
```

**Двоичные адаптеры (для примитивных типов вроде `int`, `bool`, `string` и `float`)**

```
BinaryReader, BinaryWriter
```

**Адаптеры XML (рассматривались в главе 11)**

```
XmlReader, XmlWriter
```

Отношения между упомянутыми типами представлены на рис. 15.5.

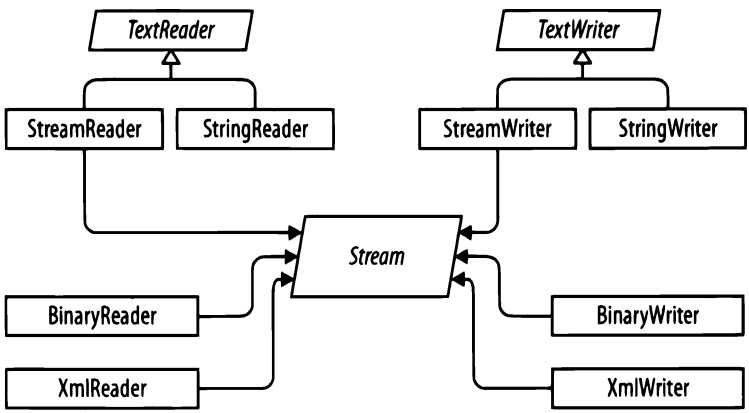



***Рис. 15.5.*** *Средства чтения и записи*

## Текстовые адаптеры

Типы `TextReader` и `TextWriter` являются абстрактными базовыми классами для адаптеров, которые имеют дело исключительно с символами и строками. С каждым из них в .NET Framework связаны две универсальные реализации.

- **`StreamReader/StreamWriter`.** Применяют для своего хранилища низкоуровневых данных класс `Stream`, транслируя байты потока в символы или строки.
- **`StringReader/StringWriter`.** Реализуют `TextReader/TextWriter`, используя строки в памяти.

В табл. 15.2 перечислены члены класса `TextReader` по категориям. Метод `Peek` возвращает следующий символ из потока, не перемещая текущую позицию вперед. Метод `Peek` и версия без аргументов метода `Read` возвращают –1, если встречается конец потока, и целочисленное значение, которое может быть приведено непосредственно к типу `char`, в противном случае.

Перегруженная версия Read, принимающая буфер char[], идентична по функциональности методу ReadBlock. Метод ReadLine производит чтение до тех пор, пока не встретит в последовательности <CR> (символ 13), <LF> (символ 10) или пару <CR+LF>. Затем он возвращает строку с отброшенными символами <CR>/<LF>.

**Таблица 15.2. Члены класса TextReader**

| Категория | Члены |
|---|---|
| Чтение одного символа | `public virtual int Peek(); // Результат приводится к char`<br>`public virtual int Read(); // Результат приводится к char` |
| Чтение множества символов | `public virtual int Read (char[] buffer, int index, int count);`<br>`public virtual int ReadBlock (char[] buffer, int index, int count);`<br>`public virtual string ReadLine();`<br>`public virtual string ReadToEnd();` |
| Закрытие | `public virtual void Close();`<br>`public void Dispose(); // То же, что и Close` |
| Другие | `public static readonly TextReader Null;`<br>`public static TextReader Synchronized (TextReader reader);` |

Последовательность новой строки в Windows приближенно моделирует механическую пишущую машинку: возврат каретки (символ 13), за которым следует перевод строки (символ 10). Соответствующая строка C# выглядит как "\r\n". Изменение порядка следования символов на обратный приведет к получению либо двух новых строк, либо вообще ни одной!

Класс TextWriter имеет аналогичные методы для записи (табл. 15.3). Методы Write и WriteLine дополнительно перегружены, чтобы принимать каждый примитивный тип плюс тип object. Такие методы просто вызывают метод ToString на том, что им передается (возможно, через реализацию интерфейса IFormatProvider, указанную или при вызове метода, или при конструировании экземпляра TextWriter).

**Таблица 15.3. Члены класса TextWriter**

| Категория | Члены |
|---|---|
| Запись одного символа | `public virtual void Write (char value);` |
| Запись множества символов | `public virtual void Write (string value);`<br>`public virtual void Write (char[] buffer, int index, int count);`<br>`public virtual void Write (string format, params object[] arg);`<br>`public virtual void WriteLine (string value);` |
| Закрытие и сбрасывание | `public virtual void Close();`<br>`public void Dispose(); // То же, что и Close`<br>`public virtual void Flush();` |
| Форматирование и кодирование | `public virtual IFormatProvider FormatProvider { get; }`<br>`public virtual string NewLine { get; set; }`<br>`public abstract Encoding Encoding { get; }` |
| Другие | `public static readonly TextWriter Null;`<br>`public static TextWriter Synchronized (TextWriter writer);` |

Метод WriteLine просто дополняет заданный текст последовательностью <CR+LF>, что можно изменить с помощью свойства NewLine (полезно для взаимодействия с файлами в форматах Unix).

Подобно Stream классы TextReader и TextWriter предлагают для своих методов чтения/записи асинхронные версии на основе задач.

### StreamReader и StreamWriter

В следующем примере экземпляр StreamWriter записывает две строки текста в файл, и затем экземпляр StreamReader производит чтение из этого файла:

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}

using (FileStream fs = File.OpenRead ("test.txt"))
using (TextReader reader = new StreamReader (fs))
{
  Console.WriteLine (reader.ReadLine());    // Line1
  Console.WriteLine (reader.ReadLine());    // Line2
}
```

Поскольку текстовые адаптеры настолько часто связываются с файлами, для сокращения объема кода класс File предоставляет статические методы CreateText, AppendText и OpenText:

```
using (TextWriter writer = File.CreateText ("test.txt"))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}

using (TextWriter writer = File.AppendText ("test.txt"))
  writer.WriteLine ("Line3");

using (TextReader reader = File.OpenText ("test.txt"))
  while (reader.Peek() > -1)
    Console.WriteLine (reader.ReadLine());  // Line1
                                            // Line2
                                            // Line3
```

Здесь еще иллюстрируется, как осуществлять проверку на предмет достижения конца файла (через reader.Peek()). Другой способ предполагает чтение до тех пор, пока reader.ReadLine не возвратит null.

Можно также выполнять чтение и запись других типов данных, подобных целым числам, но из-за того, что TextWriter вызывает на них метод ToString, при чтении потребуется производить разбор строки:

```
using (TextWriter w = File.CreateText ("data.txt"))
{
  w.WriteLine (123);   // Записывает "123"
  w.WriteLine (true);  // Записывает слово "true"
}
```

```
using (TextReader r = File.OpenText ("data.txt"))
{
  int myInt = int.Parse (r.ReadLine());  // myInt == 123
  bool yes = bool.Parse (r.ReadLine());  // yes == true
}
```

### Кодировки символов

Сами по себе `TextReader` и `TextWriter` — всего лишь абстрактные классы, не подключенные ни к потоку, ни к опорному хранилищу. Однако типы `StreamReader` и `StreamWriter` подключены к лежащему в основе байт-ориентированному потоку, поэтому они должны выполнять преобразование между символами и байтами. Они делают это посредством класса `Encoding` из пространства имен `System.Text`, который выбирается при конструировании экземпляра `StreamReader` или `StreamWriter`. Если ничего не выбрано, тогда применяется стандартная кодировка UTF-8.

В случае явного указания кодировки экземпляр `StreamWriter` по умолчанию будет записывать в начале потока префикс для идентификации кодировки. Обычно такое действие нежелательно и предотвратить его можно, конструируя экземпляр класса кодировки следующим образом:

```
var encoding = new UTF8Encoding (
  encoderShouldEmitUTF8Identifier:false,
  throwOnInvalidBytes:true);
```

Второй аргумент сообщает `StreamWriter` (или `StreamReader`) о необходимости генерации исключения, если встречаются байты, которые не имеют допустимой строковой трансляции для их кодировки, что соответствует стандартному поведению, когда кодировка не указана.

Простейшей из всех кодировок является ASCII, т.к. в ней каждый символ представлен одним байтом. Кодировка ASCII отображает первые 127 символов набора Unicode на одиночные байты, охватывая символы, которые находятся на англоязычной клавиатуре. Большинство других символов, включая специализированные и неанглийские символы, не могут быть представлены в ASCII и преобразуются в символ □. Стандартная кодировка UTF-8 может отобразить все выделенные символы Unicode, но она сложнее. Первые 127 символов кодируются в одиночный байт для совместимости с ASCII; остальные символы кодируются в варьирующееся количество байтов (чаще всего в два или три). Взгляните на приведенный ниже код:

```
using (TextWriter w = File.CreateText ("but.txt")) //Использовать стандартную
  w.WriteLine ("but-");                             // кодировку UTF-8.

using (Stream s = File.OpenRead ("but.txt"))
  for (int b; (b = s.ReadByte()) > -1;)
    Console.WriteLine (b);
```

За словом but выводится не стандартный знак переноса, а символ длинного тире (—), U+2014. Давайте исследуем вывод:

```
98   // b
117  // u
116  // t
226  // байт 1 длинного тире    Обратите внимание, что значения байтов
128  // байт 2 длинного тире    больше 128 для каждой части
148  // байт 3 длинного тире    многобайтной последовательности.
13   // <CR>
10   // <LF>
```

Символ длинного тире находится за пределами первых 127 символов набора Unicode и потому при кодировании в UTF-8 требует более одного байта (трех в данном случае). Кодировка UTF-8 эффективна с западным алфавитом, т.к. большинство популярных символов потребляют только один байт. Она также легко понижается до ASCII просто за счет игнорирования всех байтов со значениями больше 127. Недостаток кодировки UTF-8 в том, что поиск внутри потока является ненадежным, поскольку позиция символа не соответствует позиции его байтов в потоке. Альтернативой является кодировка UTF-16 (обозначенная просто как `Unicode` в классе `Encoding`). Ниже показано, как записать ту же самую строку с помощью UTF-16:

```
using (Stream s = File.Create ("but.txt"))
using (TextWriter w = new StreamWriter (s, Encoding.Unicode))
  w.WriteLine ("but-");

foreach (byte b in File.ReadAllBytes ("but.txt"))
  Console.WriteLine (b);
```

Вывод будет таким:

```
255  // Маркер порядка байтов 1
254  // Маркер порядка байтов 2
98   // 'b', байт 1
0    // 'b', байт 2
117  // 'u', байт 1
0    // 'u', байт 2
116  // 't', байт 1
0    // 't', байт 2
20   // '--', байт 1
32   // '--', байт 2
13   // <CR>, байт 1
0    // <CR>, байт 2
10   // <LF>, байт 1
0    // <LF>, байт 2
```

Формально кодировка UTF-16 использует 2 или 4 байта на символ (существует около миллиона выделенных или зарезервированных символов Unicode, поэтому двух байтов не всегда достаточно). Но из-за того, что тип `char` в C# сам имеет ширину только 16 битов, кодировка UTF-16 всегда будет применять в точности 2 байта на один символ `char`. В результате упрощается переход по индексу конкретного символа внутри потока.

Кодировка UTF-16 использует двухбайтный префикс для идентификации записи байтовых пар в порядке "старший байт после младшего" или "старший байт перед младшим" (первым идет менее значащий байт или более значащий байт). Применяемый по умолчанию порядок "старший байт после младшего" считается стандартным для систем на основе Windows.

### `StringReader` и `StringWriter`

Адаптеры `StringReader` и `StringWriter` вообще не содержат внутри себя поток; взамен в качестве лежащего в основе источника данных они используют строку или экземпляр `StringBuilder`. Это означает, что никакой трансляции байтов не требуется — в действительности классы `StringReader` и `StringWriter` не делают ничего такого, чего нельзя было бы достигнуть с помощью строки или экземпляра `StringBuilder` в паре с индексной переменной. Тем не менее, их преимущество — разделение базового класса с классами `StreamReader/StreamWriter`. Например, пусть

имеется строка, содержащая XML-код, и нужно разобрать ее с помощью XmlReader. Метод XmlReader.Create принимает один из следующих аргументов:

- URI
- Stream
- TextReader

Так каким же образом выполнить XML-разбор строки? Нам повезло, потому что StringReader является подклассом TextReader. Мы можем создать экземпляр StringReader и передать его методу XmlReader.Create:

```
XmlReader r = XmlReader.Create (new StringReader (myString));
```

## Двоичные адаптеры

Классы BinaryReader и BinaryWriter осуществляют чтение и запись собственных типов данных: bool, byte, char, decimal, float, double, short, int, long, sbyte, ushort, uint и ulong, а также строк и массивов примитивных типов данных.

В отличие от StreamReader и StreamWriter двоичные адаптеры эффективно сохраняют данные примитивных типов, потому что они представлены в памяти. Таким образом, int занимает 4 байта, а double — 8 байтов. Строки записываются посредством текстовой кодировки (как в случае StreamReader и StreamWriter), но с префиксами длины, чтобы сделать возможным чтение последовательности строк без необходимости в наличии специальных разделителей.

Предположим, что есть простой тип со следующим определением:

```
public class Person
{
  public string Name;
  public int    Age;
  public double Height;
}
```

Применяя двоичные адаптеры, в класс Person можно добавить методы для сохранения/загрузки его данных в/из потока:

```
public void SaveData (Stream s)
{
  var w = new BinaryWriter (s);
  w.Write (Name);
  w.Write (Age);
  w.Write (Height);
  w.Flush();     // Удостовериться, что буфер BinaryWriter очищен.
                 // Мы не будем освобождать/закрывать его, поэтому
}                // в поток можно записывать другие данные.
public void LoadData (Stream s)
{
  var r = new BinaryReader (s);
  Name   = r.ReadString();
  Age    = r.ReadInt32();
  Height = r.ReadDouble();
}
```

Класс BinaryReader может также производить чтение в байтовые массивы. Приведенный ниже код читает все содержимое потока, поддерживающего поиск:

```
byte[] data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

Такой прием более удобен, чем чтение напрямую из потока, поскольку он не требует использования цикла для гарантии того, что все данные были прочитаны.

## Закрытие и освобождение адаптеров потоков

Доступны четыре способа уничтожения адаптеров потоков.

1. Закрыть только адаптер.
2. Закрыть адаптер и затем закрыть поток.
3. (Для средств записи.) Сбросить адаптер и затем закрыть поток.
4. (Для средств чтения.) Закрыть только поток.

Для адаптеров методы `Close` и `Dispose` являются синонимичными в точности как в случае потоков.

Первый и второй варианты семантически идентичны, т.к. закрытие адаптера приводит к автоматическому закрытию лежащего в основе потока. Всякий раз, когда вы вкладываете операторы `using` друг в друга, то неявно принимаете второй вариант:

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
  writer.WriteLine ("Line");
```

Поскольку освобождение при вложении операторов `using` происходит наизнанку, сначала закрывается адаптер, а затем поток. Более того, если внутри конструктора адаптера сгенерировано исключение, то поток все равно закроется. Благодаря вложенным операторам `using` мало что может пойти не так, как было задумано.

Никогда не закрывайте поток перед закрытием или сбросом его средства записи, иначе любые данные, буферизированные в адаптере, будут утеряны.

Третий и четвертый варианты работают из-за того, что адаптеры относятся к необычной категории *необязательно* освобождаемых объектов. Примером, когда может быть принято решение не освобождать адаптер, является ситуация, при которой работа с адаптером закончена, но внутренний поток необходимо оставить открытым для последующего использования:

```
using (FileStream fs = new FileStream ("test.txt", FileMode.Create))
{
  StreamWriter writer = new StreamWriter (fs);
  writer.WriteLine ("Hello");
  writer.Flush();

  fs.Position = 0;
  Console.WriteLine (fs.ReadByte());
}
```

Здесь мы записываем в файл, изменяем позицию в потоке и читаем первый байт перед закрытием потока. Если мы освободим `StreamWriter`, тогда также закроется лежащий в основе объект `FileStream`, приводя к неудаче последующего чтения. Обязательное условие состоит в том, что мы вызываем метод `Flush` для обеспечения записи буфера `StreamWriter` во внутренний поток.

Адаптеры потоков — со своей семантикой необязательного освобождения — не реализуют расширенный шаблон освобождения, при котором финализатор вызывает метод `Dispose`. Это позволяет отброшенному адаптеру избежать автоматического освобождения при его подхвате сборщиком мусора.

Начиная с версии .NET Framework 4.5, в классах `StreamReader/StreamWriter` появился новый конструктор, который инструктирует поток о необходимости оставаться открытым после освобождения. Следовательно, мы можем переписать предыдущий пример так:

```
using (var fs = new FileStream ("test.txt", FileMode.Create))
{
  using (var writer = new StreamWriter (fs, new UTF8Encoding (false, true),
                                         0x400, true))
    writer.WriteLine ("Hello");
  fs.Position = 0;
  Console.WriteLine (fs.ReadByte());
  Console.WriteLine (fs.Length);
}
```

# Потоки со сжатием

В пространстве имен `System.IO.Compression` доступны два универсальных потока со сжатием: `DeflateStream` и `GZipStream`. Оба они применяют популярный алгоритм сжатия, подобный алгоритму, который используется при создании архивов в формате ZIP. Указанные классы отличаются тем, что `GZipStream` записывает дополнительную информацию в начале и в конце, включая код CRC для обнаружения ошибок. Вдобавок класс `GZipStream` соответствует стандарту, распознаваемому другим программным обеспечением.

Оба потока позволяют осуществлять чтение и запись со следующими оговорками:

- при сжатии вы всегда записываете в поток;
- при распаковке вы всегда читаете из потока.

Классы `DeflateStream` и `GZipStream` являются декораторами; они сжимают или распаковывают данные из другого потока, который указывается при конструировании их экземпляров. В следующем примере мы сжимаем и распаковываем последовательность байтов, применяя `FileStream` в качестве опорного хранилища:

```
using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Compress))
  for (byte i = 0; i < 100; i++)
    ds.WriteByte (i);
using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
  for (byte i = 0; i < 100; i++)
    Console.WriteLine (ds.ReadByte());  // Выводит числа от 0 до 99
```

Даже с использованием более скромного алгоритма из двух доступных сжатый файл имеет длину 241 байт, что более чем вдвое превышает оригинал! Дело в том, что сжатие плохо работает с "плотными", неповторяющимися двоичными данными в файлах (и хуже всего с шифрованными данными, которые лишены закономерности по определению). Сжатие успешно работает с большинством текстовых файлов; в рассмотренном далее примере мы сжимаем и распаковываем текстовый поток, со-

стоящий из 1000 слов, которые случайным образом выбраны из короткого предложения. Кроме того, в примере демонстрируется соединение в цепочку потока с опорным хранилищем, потока с декоратором и адаптера (как было показано на рис. 15.1 в начале главы), а также применение асинхронных методов:

```
string[] words = "The quick brown fox jumps over the lazy dog".Split();
Random rand = new Random();

using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Compress))
using (TextWriter w = new StreamWriter (ds))
  for (int i = 0; i < 1000; i++)
    await w.WriteAsync (words [rand.Next (words.Length)] + " ");
Console.WriteLine (new FileInfo ("compressed.bin").Length);  // 1073

using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
using (TextReader r = new StreamReader (ds))
  Console.Write (await r.ReadToEndAsync());  // Вывод показан ниже:

lazy lazy the fox the quick The brown fox jumps over fox over fox The
brown brown brown over brown quick fox brown dog dog lazy fox dog brown
over fox jumps lazy lazy quick The jumps fox jumps The over jumps dog...
```

В данном случае класс `DeflateStream` эффективно сжимает текст до 1073 байтов — чуть более одного байта на слово.

## Сжатие в памяти

Иногда сжатие нужно выполнять полностью в памяти. Ниже показано, как для такой цели использовать класс `MemoryStream`:

```
byte[] data = new byte[1000]; // Мы можем ожидать хороший коэффициент
                              // сжатия для пустого массива!
var ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress))
  ds.Write (data, 0, data.Length);

byte[] compressed = ms.ToArray();
Console.WriteLine (compressed.Length);  // 11

// Распаковка обратно в массив data:
ms = new MemoryStream (compressed);
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += ds.Read (data, i, 1000 - i));
```

Оператор `using` вокруг `DeflateStream` закрывает его рекомендуемым способом, сбрасывая любые незаписанные буферы. Вдобавок также закрывается внутренний поток `MemoryStream`, т.е. для извлечения данных нам придется вызвать метод `ToArray`.

Ниже представлен альтернативный подход, не закрывающий поток `MemoryStream`, в котором используются асинхронные методы чтения и записи:

```
byte[] data = new byte[1000];

MemoryStream ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress, true))
  await ds.WriteAsync (data, 0, data.Length);
Console.WriteLine (ms.Length);  // 113
ms.Position = 0;
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += await ds.ReadAsync (data, i, 1000 - i));
```

Дополнительный флаг, переданный конструктору DeflateStream, сообщает о том, что в отношении освобождения лежащего в основе потока не следует соблюдать обычный протокол. Другими словами, поток MemoryStream остается открытым, позволяя устанавливать его в нулевую позицию и читать повторно.

# Работа с ZIP-файлами

Долгожданным средством в версии .NET Framework 4.5 стала поддержка популярного формата сжатия ZIP-файлов, которая обеспечивается новыми классами ZipArchive и ZipFile из пространства имен System.IO.Compression (в сборке System.IO.Compression.FileSystem.dll). Преимущество этого формата над DeflateStream и GZipStream в том, что он действует как контейнер для множества файлов и совместим с ZIP-файлами, созданными с помощью проводника Windows или других утилит сжатия.

Класс ZipArchive работает с потоками, тогда как ZipFile используется в более распространенном сценарии работы с файлами. (ZipFile является статическим вспомогательным классом для ZipArchive.)

Метод CreateFromDirectory класса ZipFile добавляет все файлы из указанного каталога в ZIP-файл:

```
ZipFile.CreateFromDirectory (@"d:\MyFolder", @"d:\compressed.zip");
```

Метод ExtractToDirectory выполняет обратное действие, извлекая содержимое ZIP-файла в заданный каталог:

```
ZipFile.ExtractToDirectory (@"d:\compressed.zip", @"d:\MyFolder");
```

При сжатии можно выбирать оптимизацию по размеру файла или по скорости, а также необходимость включения в архив имени исходного каталога. Последняя опция приведет к тому, что в нашем примере внутри архива создается подкаталог по имени MyFolder, куда будут помещены сжатые файлы.

Класс ZipFile имеет метод Open, предназначенный для чтения/записи индивидуальных элементов. Он возвращает объект ZipArchive (который также можно получить, создав экземпляр ZipArchive с объектом Stream). При вызове метода Open потребуется указать имя файла и действие, которое должно быть произведено с архивом — Read (чтение), Create (создание) или Update (обновление). Затем можно выполнить перечисление по существующим элементам через свойство Entries или найти отдельный файл с помощью метода GetEntry:

```
using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Read))
  foreach (ZipArchiveEntry entry in zip.Entries)
    Console.WriteLine (entry.FullName + " " + entry.Length);
```

В классе ZipArchiveEntry также есть методы Delete, ExtractToFile (на самом деле он представляет собой расширяющий метод из класса ZipFileExtensions) и Open, который возвращает экземпляр Stream с возможностью чтения/записи. Создавать новые элементы можно посредством вызова метода CreateEntry (или расширяющего метода CreateEntryFromFile) на ZipArchive. Приведенный ниже код создает архив d:\zz.zip, к которому добавляется файл foo.dll со структурой каталогов bin\X86 внутри архива:

```
byte[] data = File.ReadAllBytes (@"d:\foo.dll");
using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Update))
  zip.CreateEntry (@"bin\X64\foo.dll").Open().Write (data, 0, data.Length);
```

То же самое можно было бы сделать полностью в памяти, создав экземпляр `ZipArchive` с потоком `MemoryStream`.

# Операции с файлами и каталогами

Пространство имен `System.IO` предоставляет набор типов для выполнения в отношении файлов и каталогов "обслуживающих" операций, таких как копирование и перемещение, создание каталогов и установка файловых атрибутов и прав доступа. Для большинства средств можно выбирать из двух классов: один предлагает статические методы, а другой – методы экземпляра.

**Статические классы**

`File` и `Directory`

**Классы с методами экземпляра (сконструированного с указанием имени файла или каталога)**

`FileInfo` и `DirectoryInfo`

Вдобавок имеется статический класс по имени `Path`. Он ничего не делает с файлами или каталогами, а предоставляет методы строкового манипулирования для имен файлов и путей к каталогам. Класс `Path` также помогает при работе с временными файлами.

В случае приложений UWP почитайте также раздел "Файловый ввод-вывод в UWP" далее в главе.

## Класс `File`

`File` – это статический класс, все методы которого принимают имя файла. Имя файла может или указываться относительно текущего каталога, или быть полностью определенным, включая каталог. Ниже перечислены методы класса `File` (все они являются `public` и `static`):

```
bool Exists (string path);  // Возвращает true, если файл существует

void Delete(string path);
void Copy  (string sourceFileName, string destFileName);
void Move  (string sourceFileName, string destFileName);
void Replace (string sourceFileName, string destinationFileName,
                                     string destinationBackupFileName);

FileAttributes GetAttributes (string path);
void SetAttributes           (string path, FileAttributes fileAttributes);

void Decrypt (string path);
void Encrypt (string path);

DateTime GetCreationTime   (string path);  // Также доступны
DateTime GetLastAccessTime (string path);  // версии UTC.
DateTime GetLastWriteTime  (string path);

void SetCreationTime   (string path, DateTime creationTime);
void SetLastAccessTime (string path, DateTime lastAccessTime);
void SetLastWriteTime  (string path, DateTime lastWriteTime);

FileSecurity GetAccessControl (string path);
FileSecurity GetAccessControl (string path,
                               AccessControlSections includeSections);
void SetAccessControl (string path, FileSecurity fileSecurity);
```

Метод `Move` генерирует исключение, если файл назначения уже существует; метод `Replace` этого не делает. Оба метода позволяют переименовывать файл, а также перемещать его в другой каталог.

Метод `Delete` генерирует исключение `UnauthorizedAccessException`, если файл помечен как предназначенный только для чтения; ситуацию можно выяснить заранее, вызвав метод `GetAttributes`. Метод `GetAttributes` возвращает значение перечисления `FileAttribute` со следующими членами:

```
Archive, Compressed, Device, Directory, Encrypted,
Hidden, Normal, NotContentIndexed, Offline, ReadOnly,
ReparsePoint, SparseFile, System, Temporary
```

Члены перечисления `FileAttribute` допускают комбинирование. Ниже показано, как переключить один атрибут файла, не затрагивая остальные:

```
string filePath = @"c:\temp\test.txt";

FileAttributes fa = File.GetAttributes (filePath);
if ((fa & FileAttributes.ReadOnly) != 0)
{
  // Использовать операцию исключающего ИЛИ (^) для переключения флага ReadOnly
  fa ^= FileAttributes.ReadOnly;
  File.SetAttributes (filePath, fa);
}
// Теперь можно удалить файл, например:
File.Delete (filePath);
```

Класс `FileInfo` предлагает более простой способ изменения флага доступности только для чтения, связанного с файлом:

```
new FileInfo (@"c:\temp\test.txt").IsReadOnly = false;
```

### Атрибуты сжатия и шифрования

Атрибуты файла `Compressed` и `Encrypted` соответствуют флажкам сжатия и шифрования в диалоговом окне *свойств* файла или каталога, которое можно открыть в проводнике Windows. Такой тип сжатия и шифрования *прозрачен* в том, что ОС делает всю работу "за кулисами", позволяя читать и записывать простые данные.

Для изменения атрибута `Compressed` или `Encrypted` нельзя применять метод `SetAttributes` — он молча откажет, если вы попытаетесь! В случае шифрования обойти проблему легко: нужно просто вызывать методы `Encrypt` и `Decrypt` класса `File`. В отношении сжатия ситуация сложнее; одно из решений предполагает использование API-интерфейса Windows Management Instrumentation (WMI) из пространства имен `System.Management`. Следующий метод сжимает каталог, возвращая 0 в случае успеха (или код ошибки WMI в случае неудачи):

```
static uint CompressFolder (string folder, bool recursive)
{
  string path = "Win32_Directory.Name='" + folder + "'";
  using (ManagementObject dir = new ManagementObject (path))
  using (ManagementBaseObject p = dir.GetMethodParameters ("CompressEx"))
  {
    p ["Recursive"] = recursive;
    using (ManagementBaseObject result = dir.InvokeMethod ("CompressEx",
                                                           p, null))
      return (uint) result.Properties ["ReturnValue"].Value;
  }
}
```

Для выполнения распаковки имя `CompressEx` понадобится заменить именем `UncompressEx`.

Прозрачное шифрование полагается на ключ, построенный на основе пароля пользователя, вошедшего в систему. Система устойчива к изменениям пароля, которые производятся аутентифицированным пользователем, но если пароль сбрасывается администратором, тогда данные в зашифрованных файлах восстановлению не подлежат.

Прозрачное шифрование и сжатие требуют специальной поддержки со стороны файловой системы. Файловая система NTFS (чаще всего применяемая на жестких дисках) такие возможности поддерживает, а CDFS (на компакт-дисках) и FAT (на сменных носителях) – нет.

Определить, поддерживает ли том сжатие и шифрование, можно посредством взаимодействия с Win32:

```
using System;
using System.IO;
using System.Text;
using System.ComponentModel;
using System.Runtime.InteropServices;

class SupportsCompressionEncryption
{
  const int SupportsCompression = 0x10;
  const int SupportsEncryption = 0x20000;

  [DllImport ("Kernel32.dll", SetLastError = true)]
  extern static bool GetVolumeInformation (string vol, StringBuilder name,
    int nameSize, out uint serialNum, out uint maxNameLen, out uint flags,
    StringBuilder fileSysName, int fileSysNameSize);

  static void Main()
  {
    uint serialNum, maxNameLen, flags;
    bool ok = GetVolumeInformation (@"C:\", null, 0, out serialNum,
                                    out maxNameLen, out flags, null, 0);
    if (!ok)
      throw new Win32Exception();

    bool canCompress = (flags & SupportsCompression) != 0;
    bool canEncrypt = (flags & SupportsEncryption) != 0;
  }
}
```

## Безопасность файлов

Методы `GetAccessControl` и `SetAccessControl` позволяют запрашивать и изменять права доступа ОС, назначенные пользователям и ролям, через объект `FileSecurity` (из пространства имен `System.Security.AccessControl`). Объект `FileSecurity` можно также передавать конструктору `FileStream` для указания прав доступа при создании нового файла.

В приведенном ниже примере мы выводим существующие права доступа к файлу, после чего назначаем права на выполнение группе Users:

```
using System;
using System.IO;
```

```
using System.Security.AccessControl;
using System.Security.Principal;

...

FileSecurity sec = File.GetAccessControl (@"d:\test.txt");
AuthorizationRuleCollection rules = sec.GetAccessRules (true, true,
                                                typeof (NTAccount));
foreach (FileSystemAccessRule rule in rules)
{
  Console.WriteLine (rule.AccessControlType);      // Allow или Deny
  Console.WriteLine (rule.FileSystemRights);       // Например, FullControl
  Console.WriteLine (rule.IdentityReference.Value); // Например, MyDomain/Joe
}
var sid = new SecurityIdentifier (WellKnownSidType.BuiltinUsersSid, null);
string usersAccount = sid.Translate (typeof (NTAccount)).ToString();

FileSystemAccessRule newRule = new FileSystemAccessRule
  (usersAccount, FileSystemRights.ExecuteFile, AccessControlType.Allow);

sec.AddAccessRule (newRule);
File.SetAccessControl (@"d:\test.txt", sec);
```

В разделе "Специальные папки" далее в главе будет представлен еще один пример.

## Класс `Directory`

Статический класс `Directory` предлагает набор методов, аналогичных методам в классе `File` – для проверки существования каталога (`Exists`), для перемещения каталога (`Move`), для удаления каталога (`Delete`), для получения/установки времени создания или времени последнего доступа и для получения/установки разрешений безопасности. Кроме того, класс `Directory` открывает доступ к следующим статическим методам:

```
string GetCurrentDirectory ();
void   SetCurrentDirectory (string path);

DirectoryInfo CreateDirectory (string path);
DirectoryInfo GetParent       (string path);
string        GetDirectoryRoot (string path);

string[] GetLogicalDrives();

// Все перечисленные ниже методы возвращают полные пути:

string[] GetFiles              (string path);
string[] GetDirectories        (string path);
string[] GetFileSystemEntries (string path);

IEnumerable<string> EnumerateFiles              (string path);
IEnumerable<string> EnumerateDirectories        (string path);
IEnumerable<string> EnumerateFileSystemEntries (string path);
```

Последние три метода появились в версии .NET Framework 4.0. Они потенциально более эффективны, чем методы `Get*`, т.к. к ним применяется ленивое выполнение – данные извлекаются из файловой системы при перечислении последовательности. Методы `Enumerate*` особенно хорошо подходят для запросов LINQ.

Методы `Enumerate*` и `Get*` перегружены, чтобы также принимать параметры `searchPattern` (строка) и `searchOption` (перечисление). В случае указания

SearchOption.SearchAllSubDirectories будет выполняться рекурсивный поиск в подкаталогах. Методы *FileSystemEntries комбинируют результаты методов *Files и *Directories.

Вот как создать каталог, если он еще не существует:

```
if (!Directory.Exists (@"d:\test"))
  Directory.CreateDirectory (@"d:\test");
```

## FileInfo и DirectoryInfo

Статические методы классов File и Directory удобны для выполнения одиночной операции над файлом или каталогом. Если необходимо вызвать последовательность методов подряд, то классы FileInfo и DirectoryInfo предоставляют объектную модель, которая облегчает работу.

Класс FileInfo предлагает большинство статических методов класса File в форме методов экземпляра – с несколькими дополнительными свойствами вроде Extension, Length, IsReadOnly и Directory – для возвращения объекта DirectoryInfo. Например:

```
FileInfo fi = new FileInfo (@"c:\temp\FileInfo.txt");
Console.WriteLine (fi.Exists);          // false

using (TextWriter w = fi.CreateText())
  w.Write ("Some text");

Console.WriteLine (fi.Exists);          // false (по-прежнему)
fi.Refresh();
Console.WriteLine (fi.Exists);          // true

Console.WriteLine (fi.Name);            // FileInfo.txt
Console.WriteLine (fi.FullName);        // c:\temp\FileInfo.txt
Console.WriteLine (fi.DirectoryName);   // c:\temp
Console.WriteLine (fi.Directory.Name); // temp
Console.WriteLine (fi.Extension);       // .txt
Console.WriteLine (fi.Length);          // 9

fi.Encrypt();
fi.Attributes ^= FileAttributes.Hidden;  // (Переключить флаг "скрытый")
fi.IsReadOnly = true;

Console.WriteLine (fi.Attributes);      // ReadOnly,Archive,Hidden,Encrypted
Console.WriteLine (fi.CreationTime);    // 3/09/2018 1:24:05 PM

fi.MoveTo (@"c:\temp\FileInfoX.txt");

DirectoryInfo di = fi.Directory;
Console.WriteLine (di.Name);            // temp
Console.WriteLine (di.FullName);        // c:\temp
Console.WriteLine (di.Parent.FullName); // c:\
di.CreateSubdirectory ("SubFolder");
```

А вот как использовать класс DirectoryInfo для перечисления файлов и подкаталогов:

```
DirectoryInfo di = new DirectoryInfo (@"e:\photos");

foreach (FileInfo fi in di.GetFiles ("*.jpg"))
  Console.WriteLine (fi.Name);

foreach (DirectoryInfo subDir in di.GetDirectories())
  Console.WriteLine (subDir.FullName);
```

## Path

В статическом классе Path определены методы и поля для работы с путями и именами файлов. Пусть имеются следующие определения:

```
string dir  = @"c:\mydir";
string file = "myfile.txt";
string path = @"c:\mydir\myfile.txt";

Directory.SetCurrentDirectory (@"k:\demo");
```

Ниже приведены выражения, демонстрирующие применение методов и полей класса Path.

| Выражение | Результат |
|---|---|
| `Directory.GetCurrentDirectory()` | `k:\demo\` |
| `Path.IsPathRooted(file)` | `False` |
| `Path.IsPathRooted(path)` | `True` |
| `Path.GetPathRoot(path)` | `c:\` |
| `Path.GetDirectoryName(path)` | `c:\mydir` |
| `Path.GetFileName(path)` | `myfile.txt` |
| `Path.GetFullPath(file)` | `k:\demo\myfile.txt` |
| `Path.Combine(dir, file)` | `c:\mydir\myfile.txt` |
| **Файловые расширения:** | |
| `Path.HasExtension(file)` | `True` |
| `Path.GetExtension(file)` | `.txt` |
| `Path.GetFileNameWithoutExtension(file)` | `myfile` |
| `Path.ChangeExtension(file, ".log")` | `myfile.log` |
| **Разделители и символы:** | |
| `Path.AltDirectorySeparatorChar` | `/` |
| `Path.PathSeparator` | `;` |
| `Path.VolumeSeparatorChar` | `:` |
| `Path.GetInvalidPathChars()` | символы от 0 до 31 и `"<>\|` |
| `Path.GetInvalidFileNameChars()` | символы от 0 до 31 и `"<>\|:*?\/` |
| **Временные файлы:** | |
| `Path.GetTempPath()` | *<папка локального пользователя>*`\Temp` |
| `Path.GetRandomFileName()` | `d2dwuzjf.dnp` |
| `Path.GetTempFileName()` | *<папка локального пользователя>* `\Temp\tmp14B.tmp` |

Метод Combine особенно полезен: он позволяет комбинировать каталог и имя файла – или два каталога – без предварительной проверки, присутствует ли обратная косая черта.

Метод GetFullPath преобразует путь, указанный относительно текущего каталога, в абсолютный путь. Он принимает значения, подобные `..\..\file.txt`.

Метод GetRandomFileName возвращает по-настоящему уникальное символьное имя в формате 8.3, не создавая файла. Метод GetTempFileName генерирует временное имя файла с использованием автоинкрементного счетчика, который повторяется для каждых 65 000 файлов. Затем он создает в локальном временном каталоге пустой файл с таким именем.

По завершении работы с файлом, имя которого сгенерировано методом GetTempFileName, вы должны его удалить; иначе со временем возникнет исключение (после 65 000 вызовов GetTempFileName). Если это проблематично, тогда для результатов выполнения GetTempPath и GetRandomFileName можно вызвать метод Combine. Только будьте осторожны, чтобы не переполнить жесткий диск пользователя!

## Специальные папки

В классах Path и Directory отсутствует средство нахождения таких папок, как My Documents, Program Files, Application Data и т.д. Взамен задача решается с помощью метода GetFolderPath класса System.Environment:

```
string myDocPath = Environment.GetFolderPath
  (Environment.SpecialFolder.MyDocuments);
```

Значения перечисления Environment.SpecialFolder охватывают все специальные каталоги в Windows:

```
AdminTools
ApplicationData
CDBurning
CommonAdminTools
CommonApplicationData
CommonDesktopDirectory
CommonDocuments
CommonMusic
CommonOemLinks
CommonPictures
CommonProgramFiles
CommonProgramFilesX86
CommonPrograms
CommonStartMenu
CommonStartup
CommonTemplates
CommonVideos
Cookies
Desktop
DesktopDirectory
Favorites
Fonts
History
InternetCache
LocalApplicationData
LocalizedResources
MyComputer
MyDocuments
MyMusic
MyPictures
MyVideos
NetworkShortcuts
Personal
PrinterShortcuts
ProgramFiles
ProgramFilesX86
Programs
Recent
Resources
SendTo
StartMenu
Startup
System
SystemX86
Templates
UserProfile
Windows
```

Значения `Environment.SpecialFolder` покрывают все кроме каталога .NET Framework, который можно получить следующим образом:

```
System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()
```

Особую ценность представляет каталог `ApplicationData`: именно здесь можно хранить настройки, которые перемещаются с пользователем по сети (если блуждающие профили разрешены в домене сети), а также каталог `LocalApplicationData`, предназначенный для неперемещаемых данных (специфичных для зарегистрированного пользователя), и каталог `CommonApplicationData`, который разделяется всеми пользователями компьютера. Запись данных приложения в указанные папки считается предпочтительнее применения реестра Windows. Стандартный протокол сохранения данных в этих каталогах предусматривает создание подкаталога с именем, которое совпадает с названием приложения:

```
string localAppDataPath = Path.Combine (
  Environment.GetFolderPath (Environment.SpecialFolder.ApplicationData),
  "MyCoolApplication");

if (!Directory.Exists (localAppDataPath))
  Directory.CreateDirectory (localAppDataPath);
```

При работе с каталогом `CommonApplicationData` есть одна ловушка: если пользователь запускает программу с повышенными административными полномочиями, после чего она создает папки и файлы в `CommonApplicationData`, то пользователю может не хватить полномочий для замены этих файлов позже, когда он запустит программу от имени обычной учетной записи. (Похожая проблема возникает при переключении между учетными записями с ограниченными полномочиями.) Такую проблему можно обойти за счет создания желаемой папки (с правами доступа для кого угодно) как части процесса установки. В качестве альтернативы, если запустить следующий код сразу же после создания папки в `CommonApplicationData` (перед записью любых файлов), то будет обеспечен неограниченный доступ любому члену группы Users:

```
public void AssignUsersFullControlToFolder (string path)
{
  try
  {
    var sec = Directory.GetAccessControl (path);
    if (UsersHaveFullControl (sec)) return;

    var rule = new FileSystemAccessRule (
      GetUsersAccount().ToString(),
      FileSystemRights.FullControl,
      InheritanceFlags.ContainerInherit | InheritanceFlags.ObjectInherit,
      PropagationFlags.None,
      AccessControlType.Allow);

    sec.AddAccessRule (rule);
    Directory.SetAccessControl (path, sec);
  }
  catch (UnauthorizedAccessException)
  {
    // Папка уже была создана другим пользователем
  }
}
```

```
bool UsersHaveFullControl (FileSystemSecurity sec)
{
  var usersAccount = GetUsersAccount();
  var rules = sec.GetAccessRules (true, true, typeof (NTAccount))
                 .OfType<FileSystemAccessRule>();
  return rules.Any (r =>
    r.FileSystemRights == FileSystemRights.FullControl &&
    r.AccessControlType == AccessControlType.Allow &&
    r.InheritanceFlags == (InheritanceFlags.ContainerInherit |
                           InheritanceFlags.ObjectInherit) &&
    r.IdentityReference == usersAccount);
}

NTAccount GetUsersAccount()
{
  var sid = new SecurityIdentifier (WellKnownSidType.BuiltinUsersSid, null);
  return (NTAccount)sid.Translate (typeof (NTAccount));
}
```

Еще одним местом для записи конфигурационных и журнальных файлов является базовый каталог приложения, который можно получить с помощью свойства `AppDomain.CurrentDomain.BaseDirectory`. Однако поступать подобным образом не рекомендуется, потому что ОС, вероятно, не разрешит приложению записывать в этот каталог после первоначальной установки (без повышения административных полномочий).

## Запрашивание информации о томе

Запрашивать информацию об устройствах на компьютере можно посредством класса `DriveInfo`:

```
DriveInfo c = new DriveInfo ("C");      // Запросить устройство C:.

long totalSize = c.TotalSize;           // Объем в байтах.
long freeBytes = c.TotalFreeSpace;      // Игнорирует дисковую квоту.
long freeToMe  = c.AvailableFreeSpace; // Учитывает дисковую квоту.

foreach (DriveInfo d in DriveInfo.GetDrives()) //Все определенные устройства.
{
  Console.WriteLine (d.Name);           // C:\
  Console.WriteLine (d.DriveType);      // Жесткий диск
  Console.WriteLine (d.RootDirectory); // C:\

  if (d.IsReady)      // Если устройство не готово, следующие
                      // два свойства сгенерируют исключения:
  {
    Console.WriteLine (d.VolumeLabel); // The Sea Drive
    Console.WriteLine (d.DriveFormat); // NTFS
  }
}
```

Статический метод `GetDrives` возвращает все отображенные устройства, включая приводы компакт-дисков, карты памяти и сетевые устройства. `DriveType` представляет собой перечисление со следующими значениями:

```
Unknown, NoRootDirectory, Removable, Fixed, Network, CDRom, Ram
```

## Перехват событий файловой системы

Класс `FileSystemWatcher` позволяет отслеживать действия, производимые над каталогом (и дополнительно над его подкаталогами). Класс `FileSystemWatcher` поддерживает события, которые инициируются при создании, модификации, переименовании и удалении файлов или подкаталогов, а также при изменении их атрибутов. События выдаются независимо от инициатора изменения — пользователя или процесса. Ниже приведен пример:

```
static void Main() { Watch (@"c:\temp", "*.txt", true); }

static void Watch (string path, string filter, bool includeSubDirs)
{
  using (var watcher = new FileSystemWatcher (path, filter))
  {
    watcher.Created += FileCreatedChangedDeleted;
    watcher.Changed += FileCreatedChangedDeleted;
    watcher.Deleted += FileCreatedChangedDeleted;
    watcher.Renamed += FileRenamed;
    watcher.Error   += FileError;

    watcher.IncludeSubdirectories = includeSubDirs;
    watcher.EnableRaisingEvents = true;

    // Прослушивание событий; завершение по нажатию <Enter>
    Console.WriteLine ("Listening for events - press <enter> to end");
    Console.ReadLine();
  }
  // Освобождение экземпляра FileSystemWatcher останавливает выдачу
  // дальнейших событий.
}

// Файл создан, изменен или удален
static void FileCreatedChangedDeleted (object o, FileSystemEventArgs e)
  => Console.WriteLine ("File {0} has been {1}", e.FullPath, e.ChangeType);

// Файл переименован
static void FileRenamed (object o, RenamedEventArgs e)
  => Console.WriteLine ("Renamed: {0}->{1}", e.OldFullPath, e.FullPath);

// Возникла ошибка
static void FileError (object o, ErrorEventArgs e)
  => Console.WriteLine ("Error: " + e.GetException().Message);
```

Поскольку `FileSystemWatcher` инициирует события в отдельном потоке, для кода обработки событий должен быть предусмотрен перехват исключений, чтобы предотвратить нарушение работы приложения из-за возникновения ошибки. Дополнительные сведения ищите в разделе "Обработка исключений" главы 14.

Событие `Error` не информирует об ошибках, связанных с файловой системой; взамен оно отражает факт переполнения буфера событий `FileSystemWatcher` событиями `Changed`, `Created`, `Deleted` или `Renamed`. Изменить размер буфера можно с помощью свойства `InternalBufferSize`.

Свойство `IncludeSubdirectories` применяется рекурсивно. Таким образом, если создать экземпляр `FileSystemWatcher` для `C:\` со свойством `IncludeSubdirectories`, установленным в `true`, то события будут инициироваться при изменении любого файла или каталога на всем жестком диске C:.

Ловушка, в которую можно попасть при использовании `FileSystemWatcher`, связана с открытием и чтением вновь созданных или обновленных файлов до того, как полностью завершится их наполнение или обновление. Если вы работаете совместно с каким-то другим программным обеспечением, создающим файлы, то потребуется предусмотреть некоторую стратегию по смягчению проблемы, например, создание файлов с неотслеживаемым расширением и затем их переименование после завершения записи.

# Файловый ввод-вывод в UWP

Приложения UWP ограничены в том, к каким каталогам и файлам они могут иметь доступ. Просмотреть ограничения легче всего с применением типов WinRT из пространства имен `Windows.Storage`, основными двумя классами которого являются `StorageFolder` и `StorageFile`.

В Windows Runtime для Windows 8 и 8.1 использовать классы `FileStream` или `Directory/File` было вообще невозможно. В итоге написание переносимых библиотек классов оказалось затрудненным, поэтому в UWP для Windows 10 такое ограничение было ослаблено, хотя по-прежнему действуют ограничения на то, к каким каталогам и файлам разрешен доступ.

Кроме того, обратите внимание, что UWP ожидает применения асинхронных методов (см. главу 14) для содействия построению отзывчивых пользовательских интерфейсов. Использование синхронных методов `FileStream` в потоке пользовательского интерфейса UWP приведет к генерации исключения.

## Работа с каталогами

Класс `StorageFolder` представляет каталог. Получить экземпляр `StorageFolder` можно с помощью его статического метода `GetFolderFromPathAsync`, передавая ему полный путь к папке. Однако с учетом того, что WinRT разрешает доступ к файлам только в определенных местоположениях, более простой подход предполагает получение экземпляра `StorageFolder` через класс `KnownFolders`, в котором для каждого (потенциально) разрешенного местоположения определено статическое свойство:

```
public static StorageFolder DocumentsLibrary { get; }
public static StorageFolder PicturesLibrary { get; }
public static StorageFolder MusicLibrary { get; }
public static StorageFolder VideosLibrary { get; }
```

Доступ к файлам дополнительно ограничивается с помощью того, что объявлено в манифесте пакета. В частности, приложения UWP могут иметь доступ только к тем файлам, расширения которых совпадают с объявленными ассоциациями для типов файлов.

Вдобавок свойство `Package.Current.InstalledLocation` возвращает экземпляр `StorageFolder` текущего приложения (к которому открыт доступ только для чтения).

Класс `KnownFolders` также имеет свойства для доступа к устройствам со съемными носителями и папкам домашней группы.

Класс `StorageFolder` содержит вполне ожидаемые свойства (`Name`, `Path`, `DateCreated`, `DateModified`, `Attributes` и т.д.), методы для удаления/переиме-

нования папки (DeleteAsync/RenameAsync), а также методы для получения списка файлов и подкаталогов (GetFilesAsync и GetFoldersAsync).

Как должно быть понятно по именам, методы являются асинхронными, возвращая объект, который можно преобразовать в задачу с помощью расширяющего метода AsTask или применить к нему напрямую await. В приведенном ниже коде получается список всех файлов в папке документов:

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
IReadOnlyList<StorageFile> files = await docsFolder.GetFilesAsync();
foreach (IStorageFile file in files)
  Debug.WriteLine (file.Name);
```

Метод CreateFileQueryWithOptions позволяет фильтровать по заданному расширению:

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
var queryOptions = new QueryOptions(CommonFileQuery.DefaultQuery, new[] { ".txt" });
var txtFiles = await docsFolder.CreateFileQueryWithOptions (queryOptions)
                                .GetFilesAsync();
foreach (StorageFile file in txtFiles)
  Debug.WriteLine (file.Name);
```

Класс QueryOptions предлагает свойства для дополнительного управления поиском. Например, свойство FolderDepth запрашивает рекурсивный список файлов и подкаталогов в каталоге:

```
queryOptions.FolderDepth = FolderDepth.Deep;
```

## Работа с файлами

Основным классом для работы с файлами является StorageFile. Получить его экземпляр для полного пути (к которому есть права доступа) можно с помощью статического метода StorageFile.GetFileFromPathAsync, а для относительного пути — посредством вызова метода GetFileAsync на объекте класса StorageFolder (или класса, реализующего IStorageFolder):

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
StorageFile file = await docsFolder.GetFileAsync ("foo.txt");
```

Если файл не существует, тогда в данной точке генерируется исключение FileNotFoundException.

Класс StorageFile располагает такими свойствами, как Name, Path и т.д., а также методами для работы с файлами — MoveAsync, RenameAsync, CopyAsync и DeleteAsync. Метод CopyAsync возвращает экземпляр StorageFile, соответствующий новому файлу. Вдобавок есть метод CopyAndReplaceAsync, который вместо целевого имени и папки принимает целевой объект StorageFile.

Кроме того, класс StorageFile предоставляет методы, позволяющие открывать файл для чтения/записи через потоки .NET (OpenStreamForReadAsync и OpenStreamForWriteAsync). Например, следующий код создает и записывает файл по имени test.txt внутри папки документов:

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
StorageFile file = await docsFolder.CreateFileAsync
  ("test.txt", CreationCollisionOption.ReplaceExisting);
using (Stream stream = await file.OpenStreamForWriteAsync())
using (StreamWriter writer = new StreamWriter (stream))
  await writer.WriteLineAsync ("This is a test");
```

Если не указано значение `CreationCollisionOption.ReplaceExisting` и файл уже существует, то имя файла автоматически дополняется числом, чтобы сделать его уникальным.

Приведенный далее код выполняет чтение файла `test.txt`:

```
StorageFolder docsFolder = KnownFolders.DocumentsLibrary;
StorageFile file = await docsFolder.GetFileAsync ("test.txt");

using (var stream = await file.OpenStreamForReadAsync ())
using (StreamReader reader = new StreamReader (stream))
  Debug.WriteLine (await reader.ReadToEndAsync());
```

## Изолированное хранилище в приложениях UWP

Приложения UWP также имеют доступ к закрытым папкам, которые изолированы от других приложений и могут использоваться для хранения данных, связанных с приложениями:

```
Windows.Storage.ApplicationData.Current.LocalFolder
Windows.Storage.ApplicationData.Current.RoamingFolder
Windows.Storage.ApplicationData.Current.TemporaryFolder
```

Каждое из этих статических свойств возвращает объект `StorageFolder`, который можно применять для чтения/записи и получения списка файлов, как было описано ранее.

# Размещенные в памяти файлы

Размещенные в памяти файлы поддерживают две основные функции:

- эффективный произвольный доступ к данным файла;
- возможность разделения памяти между различными процессами на одном и том же компьютере.

Типы для размещенных в памяти файлов находятся в пространстве имен `System.IO.MemoryMappedFiles` и появились в версии .NET Framework 4.0. Внутренне они работают через API-интерфейс Win32, предназначенный для размещенных в памяти файлов.

## Размещенные в памяти файлы и произвольный файловый ввод-вывод

Хотя обычный класс `FileStream` допускает произвольный файловый ввод-вывод (за счет установки свойства `Position` потока), он оптимизирован для последовательного ввода-вывода. Вот грубые эмпирические правила:

- при последовательном вводе-выводе экземпляры `FileStream` примерно в 10 раз быстрее размещенных в памяти файлов;
- при произвольном вводе-выводе размещенные в памяти файлы примерно в 10 раз быстрее экземпляров `FileStream`.

Изменение свойства `Position` экземпляра `FileStream` может занимать несколько микросекунд – и задержка будет накапливаться, когда это делается в цикле. Класс `FileStream` непригоден для многопоточного доступа, т.к. по мере чтения или записи позиция в нем изменяется.

Чтобы создать размещенный в памяти файл, выполните следующие действия.

1. Получите объект файлового потока (`FileStream`) обычным образом.
2. Создайте экземпляр класса `MemoryMappedFile`, передав его конструктору объект файлового потока.
3. Вызовите метод `CreateViewAccessor` на объекте размещенного в памяти файла.

Выполнение последнего действия приводит к получению объекта `MemoryMappedViewAccessor`, который предоставляет методы для произвольного чтения и записи простых типов, структур и массивов (более подробно об этом речь пойдет в разделе "Работа с аксессорами представлений" далее в главе).

Приведенный ниже код создает файл с одним миллионом байтов и затем использует API-интерфейс размещенных в памяти файлов для чтения и записи байта в позиции 500 000:

```
File.WriteAllBytes ("long.bin", new byte [1000000]);
using (MemoryMappedFile mmf = MemoryMappedFile.CreateFromFile ("long.bin"))
using (MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor())
{
  accessor.Write (500000, (byte) 77);
  Console.WriteLine (accessor.ReadByte (500000));  // 77
}
```

При вызове метода `CreateFromFile` можно также задавать имя размещенного в памяти файла и емкость. Указание отличного от `null` имени позволяет разделять блок памяти с другими процессами (как описано в следующем разделе); указание емкости автоматически увеличивает файл до такого значения. Вот как создать файл из 1000 байтов:

```
using (var mmf = MemoryMappedFile.CreateFromFile
                 ("long.bin", FileMode.Create, null, 1000))
  ...
```

## Размещенные в памяти файлы и разделяемая память

Размещенные в памяти файлы можно также применять в качестве средства для разделения памяти между процессами, которые функционируют на одном компьютере. Один из процессов создает блок разделяемой памяти, вызывая `MemoryMappedFile.CreateNew`, в то время как другие процессы подписываются на этот блок памяти, вызывая метод `MemoryMappedFile.OpenExisting` с тем же именем. Хотя на данный блок по-прежнему ссылаются как на размещенный в памяти "файл", он находится полностью в памяти и не имеет никаких представлений на диске.

Следующий код создает размещенный в памяти разделяемый файл из 500 байтов и записывает целочисленное значение 12345 в позицию 0:

```
using (MemoryMappedFile mmFile = MemoryMappedFile.CreateNew ("Demo", 500))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor())
{
  accessor.Write (0, 12345);
  Console.ReadLine();  // Сохранить разделяемую память действующей
                       // вплоть до нажатия пользователем <Enter>.
}
```

А показанный ниже код открывает тот же самый размещенный в памяти файл и читает из него упомянутое целочисленное значение:

```
// Это может быть запущено в отдельном EXE-файле:
using (MemoryMappedFile mmFile = MemoryMappedFile.OpenExisting ("Demo"))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor())
  Console.WriteLine (accessor.ReadInt32 (0));  // 12345
```

## Работа с аксессорами представлений

Вызов метода `CreateViewAccessor` на экземпляре `MemoryMappedFile` дает в результате аксессор представления, который позволяет выполнять чтение и запись в произвольные позиции.

Методы `Read*/Write*` принимают числовые типы, `bool` и `char`, а также массивы и структуры, которые содержат элементы или поля типов значений. Ссылочные типы — и массивы либо структуры, содержащие ссылочные типы — запрещены, поскольку они не могут отображаться на неуправляемую память. Таким образом, чтобы записать строку, ее потребуется закодировать в массив байтов:

```
byte[] data = Encoding.UTF8.GetBytes ("This is a test");
accessor.Write (0, data.Length);
accessor.WriteArray (4, data, 0, data.Length);
```

Обратите внимание, что первой записывается длина; позже это позволит выяснить, сколько байтов необходимо прочитать:

```
byte[] data = new byte [accessor.ReadInt32 (0)];
accessor.ReadArray (4, data, 0, data.Length);
Console.WriteLine (Encoding.UTF8.GetString (data)); // Выводит This is a test
```

Ниже приведен пример чтения/записи структуры:

```
struct Data { public int X, Y; }
...
var data = new Data { X = 123, Y = 456 };
accessor.Write (0, ref data);
accessor.Read (0, out data);
Console.WriteLine (data.X + " " + data.Y);          // Выводит 123 456
```

Методы `Read` и `Write` работают на удивление медленно. Намного лучшей производительности можно добиться, напрямую получая доступ к неуправляемой памяти через указатель. Следующий код продолжает предыдущий пример:

```
unsafe
{
  byte* pointer = null;
  try
  {
    accessor.SafeMemoryMappedViewHandle.AcquirePointer (ref pointer);
    int* intPointer = (int*) pointer;
    Console.WriteLine (*intPointer);  // 123
  }
  finally
  {
    if (pointer != null)
      accessor.SafeMemoryMappedViewHandle.ReleasePointer();
  }
}
```

Преимущество указателей в плане производительности еще ярче проявляется при работе с крупными структурами, т.к. они позволяют иметь дело непосредственно с низкоуровневыми данными, а не использовать методы `Read/Write` для *копирования* данных между управляемой и неуправляемой памятью. Это будет подробно рассматриваться в главе 25.

# Изолированное хранилище

Каждая программа .NET имеет доступ к локальной области хранения, которая является уникальной для данной программы и называется *изолированным хранилищем*. Изолированное хранилище удобно, когда программа не имеет доступа к стандартной файловой системе, а потому не может записывать в `ApplicationData`, `LocalApplicationData`, `CommonApplicationData`, `MyDocuments` и т.д. (см. раздел "Специальные папки" ранее в главе). Это характерно для приложений Silverlight и ClickOnce, развернутых с ограниченными разрешениями зоны "Интернет". Мы раскрываем изолированное хранилище в отдельном дополнении, которое можно загрузить на веб-сайте издательства.

# 16

# Взаимодействие с сетью

Инфраструктура .NET Framework предлагает в пространствах имен `System.Net.*` множество классов, предназначенных для организации взаимодействия через стандартные сетевые протоколы, такие как HTTP, TCP/IP и FTP. Ниже приведен краткий перечень основных компонентов:

- фасадный класс `WebClient` для простых операций загрузки/выгрузки через HTTP или FTP;
- классы `WebRequest` и `WebResponse` для низкоуровневого управления операциями HTTP или FTP на стороне клиента;
- класс `HttpClient` для работы с API-интерфейсами HTTP и веб-службами REST;
- класс `HttpListener` для реализации HTTP-сервера;
- класс `SmtpClient` для формирования и отправки почтовых сообщений через SMTP;
- класс `Dns` для преобразований между доменными именами и адресами;
- классы `TcpClient`, `UdpClient`, `TcpListener` и `Socket` для прямого доступа к транспортному и сетевому уровням.

Указанные выше типы являются частью стандарта .NET Standard 2.0, т.е. приложения UWP могут ими пользоваться (если только вы не имеете дело с более старой версией платформы UWP, которая не соответствует стандарту .NET Standard 2.0). Приложения UWP могут также работать с типами WinRT для взаимодействия по протоколам TCP и UDP из пространства имен `Windows.Networking.Sockets`, которые будут представлены в последнем разделе главы. Их преимущество заключается в поддержке асинхронного программирования.

Типы .NET, рассматриваемые в данной главе, находятся в пространствах имен `System.Net.*` и `System.IO`.

# Сетевая архитектура

На рис. 16.1 показаны типы .NET для работы с сетью и коммуникационные уровни, к которым они относятся. Большинство типов взаимодействуют с *транспортным уровнем* или с *прикладным уровнем*. Транспортный уровень определяет базовые протоколы для отправки и получения байтов (TCP и UDP), а прикладной уровень – высокоуровневые протоколы, предназначенные для конкретных применений, таких как извлечение веб-страниц (HTTP), передача файлов (FTP), отправка сообщений электронной почты (SMTP) и преобразование между доменными именами и IP-адресами (DNS).

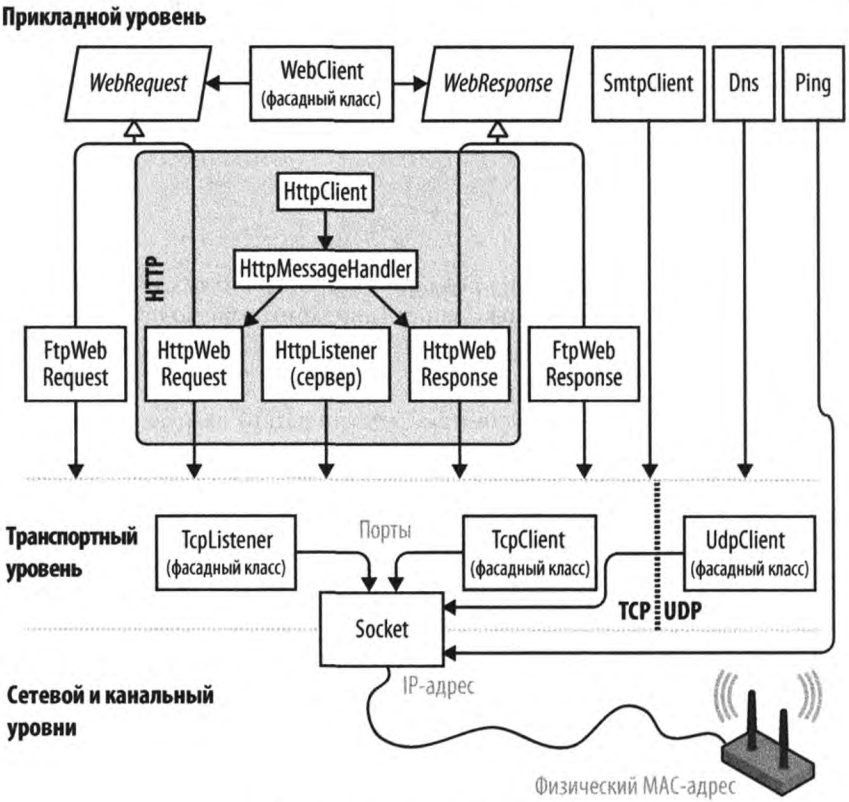



***Рис. 16.1.*** *Сетевая архитектура*

Обычно удобнее всего программировать на прикладном уровне; тем не менее, есть пара причин, по которым может требоваться работа непосредственно на транспортном уровне. Одна из них связана с необходимостью взаимодействия с прикладным протоколом, не предоставляемым .NET Framework, таким как POP3 для извлечения сообщений электронной почты. Другая причина касается реализации нестандартного протокола для специального приложения, подобного клиенту одноранговой сети.

Внутри набора прикладных протоколов особенность протокола HTTP заключается в том, что он применим к универсальным коммуникациям. Его основной режим работы – "предоставьте мне веб-страницу с заданным URL" – хорошо приспосабливается к

варианту "предоставьте мне результат обращения к этой конечной точке с заданными аргументами". (В дополнение к команде Get имеются команды Put, Post и Delete, делая возможными веб-службы на основе REST.)

Протокол HTTP также располагает богатым набором средств, которые полезны в многоуровневых бизнес-приложениях и архитектурах, ориентированных на службы. В их число входят протоколы для аутентификации и шифрования, разделение сообщений на куски, расширяемые заголовки и cookie-наборы, а также возможность совместного использования единственного порта и IP-адреса несколькими серверными приложениями. По этим причинам протокол HTTP широко поддерживается в .NET Framework – и напрямую, как описано в текущей главе, и на более высоком уровне через такие технологии, как WCF, Web Services и ASP.NET.

Инфраструктура .NET Framework обеспечивает поддержку на стороне клиента протокола FTP – популярного Интернет-протокола, предназначенного для отправки и получения файлов. Поддержка FTP на стороне сервера имеет форму IIS или серверного программного обеспечения на основе Unix.

Из предыдущего обсуждения должно быть ясно, что работа в сети представляет собой область, изобилующую аббревиатурами. Самые распространенные аббревиатуры объясняются в табл. 16.1.

**Таблица 16.1. Аббревиатуры, связанные с сетью**

| Аббревиатура | Расшифровка | Примечания |
|---|---|---|
| DNS | Domain Name Service (служба доменных имен) | Выполняет преобразования между доменными именами (скажем, `ebay.com`) и IP-адресами (например, 199.54.213.2) |
| FTP | File Transfer Protocol (протокол передачи файлов) | Основанный на Интернете протокол для отправки и получения файлов |
| HTTP | Hypertext Transfer Protocol (протокол передачи гипертекста) | Извлекает веб-страницы и запускает веб-службы |
| IIS | Internet Information Services (информационные службы Интернета) | Программное обеспечение веб-сервера производства Microsoft |
| IP | Internet Protocol (протокол Интернета) | Протокол сетевого уровня, находящийся ниже TCP и UDP |
| LAN | Local Area Network (локальная вычислительная сеть) | Большинство локальных вычислительных сетей применяют основанные на Интернете протоколы, такие как TCP/IP |
| POP | Post Office Protocol (протокол почтового офиса) | Извлекает сообщения электронной почты Интернета |
| REST | REpresentational State Transfer (передача состояния представления) | Популярная альтернатива веб-службам (Web Services), которая использует ссылки в ответах и может работать поверх базового протокола HTTP |
| SMTP | Simple Mail Transfer Protocol (простой протокол передачи почты) | Отправляет сообщения электронной почты Интернета |

*Окончание табл. 16.1*

| Аббревиатура | Расшифровка | Примечания |
|---|---|---|
| TCP | Transmission and Control Protocol (протокол управления передачей) | Интернет-протокол транспортного уровня, поверх которого построено большинство служб более высокого уровня |
| UDP | Universal Datagram Protocol (универсальный протокол передачи дейтаграмм) | Интернет-протокол транспортного уровня, применяемый для служб с низкими накладными расходами, таких как VoIP |
| UNC | Universal Naming Convention (соглашение об универсальном назначении имен) | `\\компьютер\имя_разделяемого_ресурса\имя_файла` |
| URI | Uniform Resource Identifier (универсальный идентификатор ресурса) | Вездесущая система именования ресурсов (например, `http://www.amazon.com` или `mailto:joe@bloggs.org`) |
| URL | Uniform Resource Locator (унифицированный указатель ресурса) | Формальный смысл (используется редко): подмножество URI; популярный смысл: синоним URI |

# Адреса и порты

Для функционирования коммуникаций компьютер или устройство должно иметь адрес. В Интернете применяются две системы адресации.

- **IPv4.** В настоящее время является доминирующей системой адресации; адреса IPv4 имеют ширину 32 бита. В строковом формате адреса IPv4 записываются в виде четырех десятичных чисел, разделенных точками (например, 101.102.103.104). Адрес может быть уникальным в мире или внутри отдельной *подсети* (такой как корпоративная сеть).
- **IPv6.** Более новая система 128-битной адресации. В строковом формате адреса IPv6 записываются в виде шестнадцатеричных чисел, разделенных двоеточиями (например, [3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]). Инфраструктура .NET Framework требует помещения адреса в квадратные скобки.

Класс `IPAddress` из пространства имен `System.Net` представляет адрес в обоих протоколах. Он имеет конструктор, принимающий байтовый массив, и статический метод `Parse`, который принимает корректно сформатированную строку:

```
IPAddress a1 = new IPAddress (new byte[] { 101, 102, 103, 104 });
IPAddress a2 = IPAddress.Parse ("101.102.103.104");
Console.WriteLine (a1.Equals (a2));     // True
Console.WriteLine (a1.AddressFamily);  // InterNetwork

IPAddress a3 = IPAddress.Parse
  ("[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]");
Console.WriteLine (a3.AddressFamily);  // InterNetworkV6
```

Протоколы TCP и UDP развертывают каждый IP-адрес на 65 535 портов, позволяя компьютеру с единственным адресом запускать множество приложений, каждое на своем порту. Многие приложения имеют стандартные назначения портов; скажем, протокол HTTP использует порт 80, а SMTP – порт 25.

Порты TCP и UDP с номерами от 49152 до 65535 официально свободны, поэтому они хорошо подходят для тестирования и небольших развертываний.

Комбинация IP-адреса и порта представлена в .NET Framework классом `IPEndPoint`:

```
IPAddress a = IPAddress.Parse ("101.102.103.104");
IPEndPoint ep = new IPEndPoint (a, 222);  // Порт 222
Console.WriteLine (ep.ToString());        // 101.102.103.104:222
```

Брандмауэры блокируют порты. Во многих корпоративных средах открыто лишь несколько портов — обычно порт 80 (для нешифрованного HTTP) и порт 443 (для защищенного HTTP).

# Идентификаторы URI

Идентификатор URI представляет собой особым образом сформатированную строку, которая описывает ресурс в Интернете или локальной сети, такой как веб-страница, файл или адрес электронной почты. Примерами могут служить `http://www.ietf.org`, `ftp://myisp/doc.txt` и `mailto:joe@bloggs.com`. Точный формат определен IETF (Internet Engineering Task Force — инженерная группа по развитию Интернета; `http://www.ietf.org/`).

Идентификатор URI может быть разбит на последовательность элементов, обычно включающую *схему*, *источник* и *путь*. Такое разделение осуществляет класс `Uri` из пространства имен `System`, в котором определены свойства для всех упомянутых элементов. На рис. 16.2 приведена соответствующая иллюстрация.

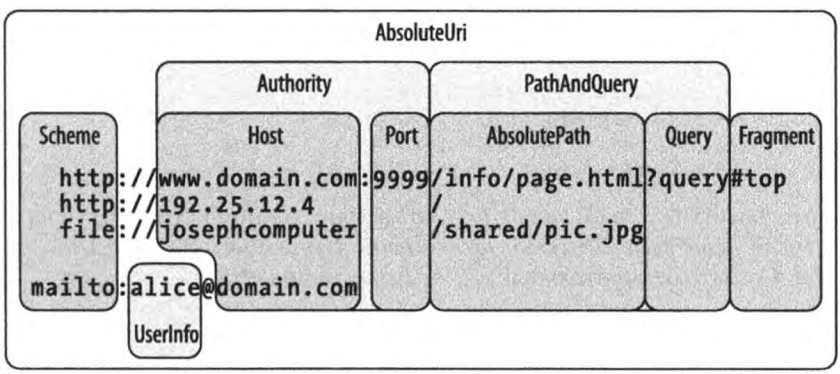



***Рис. 16.2.** Свойства класса `Uri`*

Класс `Uri` полезен, когда нужно проверить правильность формата строки URI или разбить URI на компоненты. По-другому URI можно трактовать просто как строку — большинство методов, работающих с сетью, перегружены для приема либо объекта `Uri`, либо строки.

Для создания объекта `Uri` конструктору можно передать любую из перечисленных ниже строк:

- строка URI, такая как `http://www.ebay.com` или `file://janespc/sharedpics/dolphin.jpg`;
- абсолютный путь к файлу на жестком диске вроде `c:\myfiles\data.xls`;
- путь UNC к файлу в локальной сети наподобие `\\janespc\sharedpics\dolphin.jpg`.

Путь к файлу и путь UNC автоматически преобразуются в идентификаторы URI: добавляется протокол `file:`, а символы обратной косой черты заменяются символами обычной косой черты. Перед созданием объекта `Uri` конструкторы `Uri` также выполняют некоторую базовую очистку строки, включая приведение схемы и имени хоста к нижнему регистру и удаление стандартных и пустых номеров портов. В случае указания строки URI без схемы, скажем, `www.test.com`, генерируется исключение `UriFormatException`.

Класс `Uri` имеет свойство `IsLoopback`, которое указывает, ссылается ли `Uri` на локальный хост (с IP-адресом 127.0.0.1), и свойство `IsFile`, которое указывает, ссылается ли `Uri` на локальный путь или путь UNC (`IsUnc`). Если `IsFile` равно `true`, тогда свойство `LocalPath` возвращает версию `AbsolutePath`, дружественную к локальной операционной системе (с символами обратной косой черты) и допускающую вызов метода `File.Open`.

Экземпляры `Uri` имеют свойства, предназначенные только для чтения. Чтобы модифицировать существующий `Uri`, необходимо создать объект `UriBuilder` — он имеет записываемые свойства и может быть преобразован обратно через свойство `Uri`.

Класс `Uri` также предоставляет методы для сравнения и вычитания путей:

```
Uri info = new Uri ("http://www.domain.com:80/info/");
Uri page = new Uri ("http://www.domain.com/info/page.html");

Console.WriteLine (info.Host); // www.domain.com
Console.WriteLine (info.Port); // 80
Console.WriteLine (page.Port); //80 (классу Uri известен стандартный порт HTTP)

Console.WriteLine (info.IsBaseOf (page));   // True
Uri relative = info.MakeRelativeUri (page);
Console.WriteLine (relative.IsAbsoluteUri);// False
Console.WriteLine (relative.ToString());    // page.html
```

Относительный `Uri`, такой как `page.html` в приведенном примере, сгенерирует исключение, если будет вызвано любое другое свойство или метод кроме `IsAbsoluteUri` и `ToString`. Создать относительный `Uri` напрямую можно так:

```
Uri u = new Uri ("page.html", UriKind.Relative);
```

Завершающий символ обратной косой черты в URI важен и вносит отличие в то, каким образом сервер обрабатывает запрос, если присутствует компонент пути.

Например, для URI вида `http://www.albahari.com/nutshell/` можно предположить, что веб-сервер HTTP будет искать подкаталог `nutshell` в веб-папке сайта и возвратит стандартный документ (обычно `index.html`).

Когда завершающий символ обратной косой черты отсутствует, веб-сервер будет искать файл по имени `nutshell` (без расширения) прямо в корневой папке сайта — обычно это не то, что нужно. Если такой файл не существует, то большинство веб-серверов будут считать, что пользователь допустил опечатку и возвратят ошибку 301 *Permanent Redirect* (постоянное перенаправление), предлагая клиенту повторить попытку с завершающим символом обратной косой черты. По умолчанию HTTP-клиент .NET будет прозрачно реагировать на ошибку 301 тем же способом, что и веб-браузер — повторяя попытку с предложенным URI. Это значит, что если вы опустите завершающий символ обратной косой черты, когда он должен быть включен, то запрос все равно будет работать, но потребует излишнего двухстороннего обмена.

Класс `Uri` также предлагает статические вспомогательные методы вроде `EscapeUriString`, который преобразует строку в допустимый URL, заменяя все символы с ASCII-кодами больше 127 их шестнадцатеричными представлениями. Методы `CheckHostName` и `CheckSchemeName` принимают строку и проверяют, является ли она синтаксически правильной для заданного свойства (хотя они не пытаются определить, существует ли указанный хост или URI).

# Классы клиентской стороны

Типы `WebRequest` и `WebResponse` являются общими базовыми классами, предназначенными для управления работой протоколов HTTP и FTP на стороне клиента, а также протокола `file:`. Они инкапсулируют модель "запрос/ответ", разделяемую всеми указанными протоколами: клиент делает запрос и затем ожидает ответа от сервера.

Тип `WebClient` представляет собой удобный фасадный класс, который обеспечивает обращение к классам `WebRequest` и `WebResponse`, сокращая объем подлежащего написанию кода. Класс `WebClient` позволяет выбирать, с чем иметь дело — со строками, байтовыми массивами, файлами или потоками; классы `WebRequest` и `WebResponse` поддерживают только потоки. К сожалению, полагаться целиком на `WebClient` нельзя, потому что он не поддерживает некоторые средства (такие как cookie-наборы).

Тип `HttpClient` — еще один класс, построенный на базе `WebRequest` и `WebResponse` (точнее, на `HttpWebRequest` и `HttpWebResponse`), который появился в версии .NET Framework 4.5. Класс `WebClient` действует главным образом как тонкий слой над классами запроса/ответа, а класс `HttpClient` добавляет функциональность, помогающую работать с API-интерфейсами HTTP, веб-службами REST и специальными схемами аутентификации.

Для простой загрузки/выгрузки файла, строки или байтового массива подходит и `WebClient`, и `HttpClient`. Оба класса имеют асинхронные методы, хотя выдачу информации о ходе работ поддерживает только класс `WebClient`.

По умолчанию среда CLR регулирует параллелизм HTTP. Если вы планируете применять асинхронные методы или многопоточность, чтобы делать более двух запросов за раз (через `WebRequest`, `WebClient` или `HttpClient`), то должны будете сначала увеличить предел параллелизма через статическое свойство `ServicePointManager.DefaultConnectionLimit`. По данной теме в MSDN доступна полезная статья: `http://tinyurl.com/44axxby`.

## WebClient

Ниже перечислены действия, которые понадобится выполнить для использования WebClient.

1. Создайте объект WebClient.
2. Установите свойство Proxy.
3. Установите свойство Credentials, если требуется аутентификация.
4. Вызовите метод Download*XXX* или Upload*XXX* с желаемым URI.

Класс WebClient имеет следующие методы для загрузки:

```
public void   DownloadFile    (string address, string fileName);
public string DownloadString (string address);
public byte[] DownloadData    (string address);
public Stream OpenRead        (string address);
```

Каждый из них перегружен с целью принятия объекта Uri вместо строкового адреса. Методы выгрузки похожи; их возвращаемые значения содержат ответ от сервера (если он есть):

```
public byte[] UploadFile   (string address, string fileName);
public byte[] UploadFile   (string address, string method, string fileName);
public string UploadString (string address, string data);
public string UploadString (string address, string method, string data);
public byte[] UploadData   (string address, byte[] data);
public byte[] UploadData   (string address, string method, byte[] data);
public byte[] UploadValues (string address, NameValueCollection data);
public byte[] UploadValues (string address, string method,
                                  NameValueCollection data);
public Stream OpenWrite    (string address);
public Stream OpenWrite    (string address, string method);
```

Методы UploadValues могут применяться для отправки значений HTTP-формы с аргументом method, установленным в POST. Класс WebClient также имеет свойство BaseAddress; оно позволяет указывать строку, предваряющую все адреса, такую как http://www.mysite.com/data/.

Ниже показано, как загрузить страницу с примерами кода для данной книги в файл внутри текущего каталога и затем отобразить ее в стандартном веб-браузере:

```
WebClient wc = new WebClient { Proxy = null };
wc.DownloadFile ("http://www.albahari.com/nutshell/code.aspx", "code.htm");
System.Diagnostics.Process.Start ("code.htm");
```

Класс WebClient реализует интерфейс IDisposable по *принуждению* — в силу того, что он является производным от класса Component (это позволяет ему располагаться в панели компонентов визуального редактора Visual Studio). Тем не менее, его метод Dispose во время выполнения не делает ничего полезного, а потому освобождать экземпляры WebClient не требуется.

Начиная с версии .NET Framework 4.5, класс WebClient предоставляет *асинхронные* версии для своих длительно выполняющихся методов (см. главу 14), которые возвращают задачи, пригодные для ожидания:

```
await wc.DownloadFileTaskAsync ("http://oreilly.com", "webpage.htm");
```

(Суффикс TaskAsync разрешает неоднозначность между этими методами и старыми асинхронными методами, основанными на шаблоне EAP, в которых используется суффикс Async.) К сожалению, новые методы не поддерживают стандартный шаблон TAP, регламентирующий отмену и сообщение о ходе работ. По указанной причине для отмены потребуется вызывать метод CancelAsync на объекте WebClient, а для обеспечения выдачи сведений о ходе работ – обрабатывать события DownloadProgressChanged/UploadProgressChanged. Приведенный далее код загружает веб-страницу с сообщением о ходе работ, отменяя загрузку, если она занимает более 5 секунд:

```
var wc = new WebClient();
wc.DownloadProgressChanged += (sender, args) =>
  Console.WriteLine (args.ProgressPercentage + "% complete");
Task.Delay (5000).ContinueWith (ant => wc.CancelAsync());
await wc.DownloadFileTaskAsync ("http://oreilly.com", "webpage.htm");
```

Когда запрос отменен, генерируется исключение WebException со свойством Status, установленным в WebExceptionStatus.RequestCanceled. (По историческим причинам исключение OperationCanceledException не генерируется.)

События, связанные с ходом работ, захватываются и отправляются активному контексту синхронизации, так что их обработчики могут обновлять элементы управления пользовательского интерфейса, не нуждаясь в вызове метода Dispatcher.BeginInvoke.

Если планируется поддержка отмены или сообщения о ходе работ, тогда следует избегать применения того же самого объекта WebClient для выполнения более одной операции последовательно, т.к. в итоге могут возникнуть условия состязаний.

## WebRequest и WebResponse

Классы WebRequest и WebResponse хоть и сложнее в использовании, чем WebClient, но также обладают большей гибкостью. Ниже описано, как приступить к работе с ними.

1. Вызовите метод WebRequest.Create с URI, чтобы создать объект веб-запроса.
2. Установите свойство Proxy.
3. Установите свойство Credentials, если требуется аутентификация.

Для выгрузки данных выполните следующее действие.

1. Вызовите метод GetRequestStream на объекте запроса и затем начинайте запись в поток. Если ожидается ответ, то перейдите к шагу 5.

Для загрузки данных выполните следующие действия.

1. Вызовите метод GetResponse на объекте запроса для создания объекта веб-ответа.
2. Вызовите метод GetResponseStream на объекте ответа и затем начинайте чтение из потока (здесь может помочь класс StreamReader).

Показанный далее код загружает и отображает веб-страницу с примерами кода (это переписанный предшествующий пример):

```
WebRequest req = WebRequest.Create
               ("http://www.albahari.com/nutshell/code.html");
req.Proxy = null;
using (WebResponse res = req.GetResponse())
using (Stream rs = res.GetResponseStream())
using (FileStream fs = File.Create ("code.html"))
  rs.CopyTo (fs);
```

А вот его асинхронный эквивалент:

```
WebRequest req = WebRequest.Create
               ("http://www.albahari.com/nutshell/code.html");
req.Proxy = null;
using (WebResponse res = await req.GetResponseAsync())
using (Stream rs = res.GetResponseStream())
using (FileStream fs = File.Create ("code.html"))
  await rs.CopyToAsync (fs);
```

Объект веб-ответа имеет свойство `ContentLength`, которое указывает длину потока ответа в байтах, сообщаемую сервером. Это значение поступает из заголовков ответа и может быть недоступным или некорректным. В частности, если HTTP-сервер функционирует в режиме разделения длинных ответов на куски меньших размеров, то значение `ContentLength` обычно равно -1. То же самое применимо к динамически генерируемым страницам.

Статический метод `Create` создает экземпляр подкласса типа `WebRequest`, такого как `HttpWebRequest` или `FtpWebRequest`. Выбор подкласса зависит от префикса URI (табл. 16.2).

**Таблица 16.2. Префиксы URI и типы веб-запросов**

| Префикс | Тип веб-запроса |
|---|---|
| `http:` или `https:` | `HttpWebRequest` |
| `ftp:` | `FtpWebRequest` |
| `file:` | `FileWebRequest` |

Приведение объекта веб-запроса к его конкретному типу (`HttpWebRequest` или `FtpWebRequest`) позволяет получить доступ к возможностям, специфичным для реализуемого им протокола.

Можно также зарегистрировать собственные префиксы, вызвав метод `WebRequest.RegisterPrefix`. Данный метод требует передачи префикса наряду с фабричным объектом, имеющим метод `Create`, который создает подходящий объект веб-запроса.

Протокол `https:` предназначен для защищенного (шифрованного) HTTP через SSL (Secure Sockets Layer – уровень защищенных сокетов). Встретив такой префикс, объекты `WebClient` и `WebRequest` прозрачным образом активизируют SSL (как показано в подразделе "SSL" внутри раздела "Работа с протоколом HTTP" далее в главе).

Протокол `file:` просто переадресовывает запросы объекту `FileStream`. Его целью является обеспечение согласованного протокола для чтения идентификатора URI, будь он веб-страницей, FTP-сайтом или путем к файлу.

Класс `WebRequest` имеет свойство `Timeout`, указывающее тайм-аут в миллисекундах. Когда возникает тайм-аут, генерируется исключение `WebException` со свойством `Status`, установленным в `WebExceptionStatus.Timeout`. Стандартный тайм-аут составляет 100 секунд для HTTP и бесконечность для FTP.

Объект `WebRequest` не может использоваться для множества запросов — каждый экземпляр пригоден для выполнения только одной работы.

## HttpClient

Класс `HttpClient` появился в версии .NET Framework 4.5 и предоставляет еще один уровень поверх `HttpWebRequest` и `HttpWebResponse`. Он был реализован в ответ на развитие API-интерфейсов, предназначенных для взаимодействия с протоколом HTTP и веб-службами REST, чтобы предложить улучшенный по сравнению с `WebClient` стиль работы с протоколами, который выходит за рамки простого извлечения веб-страниц. Ниже описаны основные особенности класса `HttpClient`.

- Одиночный экземпляр `HttpClient` поддерживает параллельные запросы. Чтобы добиться параллелизма с помощью класса `WebClient`, его новые экземпляры придется создавать для каждого параллельного запроса, а это станет затруднительным в случае ввода специальных заголовков, cookie-наборов и схем аутентификации.
- Класс `HttpClient` позволяет создавать и подключать специальные обработчики сообщений. Это делает возможными имитацию для модульного тестирования и построение специальных конвейеров (для регистрации в журнале, сжатия, шифрования и т.д.). Написание кода модульного тестирования, который обращается к `WebClient` — довольно болезненное занятие.
- Класс `HttpClient` имеет развитую и расширяемую систему типов для заголовков и содержимого.

Класс `HttpClient` не является полной заменой `WebClient`, поскольку он не поддерживает сообщение о ходе работ. Еще одно преимущество класса `WebClient` связано с тем, что он поддерживает протоколы FTP, `file://` и специальные схемы URI. Кроме того, он доступен в предшествующих версиях .NET Framework.

Простейший способ применения класса `HttpClient` предусматривает создание его экземпляра и вызов одного из методов `Get*` с передачей ему URI:

```
string html = await new HttpClient().GetStringAsync ("http://linqpad.net");
```

(Доступны также методы `GetByteArrayAsync` и `GetStreamAsync`.) Все методы с интенсивным вводом-выводом в классе `HttpClient` являются асинхронными (их синхронные эквиваленты отсутствуют).

В отличие от класса `WebClient` для достижения более высокой производительности при работе с `HttpClient` вы *должны* повторно использовать тот же самый экземпляр (иначе могут повториться излишние действия вроде распознавания DNS). Класс `HttpClient` разрешает параллельные операции, поэтому следующий код допустим; в нем загружаются две веб-страницы за раз:

```
var client = new HttpClient();
var task1 = client.GetStringAsync ("http://www.linqpad.net");
var task2 = client.GetStringAsync ("http://www.albahari.com");
Console.WriteLine (await task1);
Console.WriteLine (await task2);
```

В классе HttpClient имеется свойство Timeout и свойство BaseAddress, которое добавляется в качестве префикса к URI каждого запроса. В определенной степени класс HttpClient является тонкой оболочкой: большинство других свойств, которые можно в нем обнаружить, определены в другом классе по имени HttpClientHandler. Для доступа к этому классу необходимо создать и передать его экземпляр конструктору класса HttpClient:

```
var handler = new HttpClientHandler { UseProxy = false };
var client = new HttpClient (handler);
...
```

В показанном примере мы сообщаем обработчику о необходимости отключения поддержки прокси-сервера. Предусмотрены также свойства для управления cookie-наборами, автоматическим перенаправлением, аутентификацией и т.д. (мы рассмотрим их в последующих разделах и в разделе "Работа с протоколом HTTP" далее в главе).

### Метод GetAsync и сообщения ответов

Методы GetStringAsync, GetByteArrayAsync и GetStreamAsync представляют собой удобные сокращения для вызова более общего метода GetAsync, который возвращает *сообщение ответа*:

```
var client = new HttpClient();
// Метод GetAsync также принимает объект CancellationToken.
HttpResponseMessage response = await client.GetAsync ("http://...");
response.EnsureSuccessStatusCode();
string html = await response.Content.ReadAsStringAsync();
```

Класс HttpResponseMessage имеет свойства для доступа к заголовкам (см. раздел "Работа с протоколом HTTP" далее в главе) и коду состояния HTTP (StatusCode). В отличие от класса WebClient код состояния неудачи, такой как 404 (не найдено), не приводит к генерации исключения, если только не будет явно вызван метод EnsureSuccessStatusCode. Тем не менее, ошибки коммуникаций или DNS вызывают генерацию исключений (они описаны в разделе "Обработка исключений" далее в главе).

Класс HttpResponseMessage располагает методом CopyToAsync для записи в другой поток, который удобен для сохранения вывода в файле:

```
using (var fileStream = File.Create ("linqpad.html"))
  await response.Content.CopyToAsync (fileStream);
```

GetAsync является одним из четырех методов, соответствующих четырем командам HTTP (остальные методы — это PostAsync, PutAsync и DeleteAsync). Мы продемонстрируем применение метода PostAsync в разделе "Выгрузка данных формы" далее в главе.

### Метод SendAsync и сообщения запросов

Только что описанные четыре метода выступают в качестве сокращений для вызова метода SendAsync — единственного низкоуровневого метода, который делает всю работу. Сначала конструируется объект HttpRequestMessage:

```
var client = new HttpClient();
var request = new HttpRequestMessage (HttpMethod.Get, "http://...");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
...
```

Создание объекта HttpRequestMessage означает возможность настройки свойств запроса, таких как заголовки (см. раздел "Заголовки" далее в главе), и самого содержимого, позволяя выгружать данные.

### Выгрузка данных и HttpContent

После создания объекта HttpRequestMessage можно выгружать содержимое, устанавливая его свойство Content. Типом этого свойства является абстрактный класс по имени HttpContent. Инфраструктура .NET Framework включает следующие конкретные подклассы для различных видов содержимого (можно также построить собственный такой подкласс):

- ByteArrayContent
- StreamContent
- FormUrlEncodedContent (см. раздел "Выгрузка данных формы" далее в главе)
- StreamContent

Например:

```
var client = new HttpClient (new HttpClientHandler { UseProxy = false });
var request = new HttpRequestMessage (
  HttpMethod.Post, "http://www.albahari.com/EchoPost.aspx");
request.Content = new StringContent ("This is a test");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

### HttpMessageHandler

Ранее мы упоминали, что большинство свойств для настройки запросов определено не в HttpClient, а в классе HttpClientHandler. Последний в действительности представляет собой подкласс абстрактного класса HttpMessageHandler, определенного следующим образом:

```
public abstract class HttpMessageHandler : IDisposable
{
  protected internal abstract Task<HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken);

  public void Dispose();
  protected virtual void Dispose (bool disposing);
}
```

Метод SendAsync вызывается внутри метода SendAsync класса HttpClient.

Из HttpMessageHandler довольно легко создавать подклассы, и он является точкой расширения для HttpClient.

### Модульное тестирование и имитация

Подкласс HttpMessageHandler можно создавать для построения *имитированного* обработчика, который помогает проводить модульное тестирование:

```
class MockHandler : HttpMessageHandler
{
  Func <HttpRequestMessage, HttpResponseMessage> _responseGenerator;

  public MockHandler
    (Func <HttpRequestMessage, HttpResponseMessage> responseGenerator)
  {
    _responseGenerator = responseGenerator;
  }

  protected override Task <HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken)
  {
    cancellationToken.ThrowIfCancellationRequested();
    var response = _responseGenerator (request);
    response.RequestMessage = request;
    return Task.FromResult (response);
  }
}
```

Его конструктор принимает функцию, которая сообщает имитированному объекту, каким образом генерировать ответ из запроса. Такой подход наиболее универсален, потому что один и тот же обработчик способен тестировать множество запросов.

По причине вызова `Task.FromResult` метод `SendAsynch` синхронен. Мы могли бы поддерживать асинхронность, обеспечив возвращение генератором ответов объекта типа `Task<HttpResponseMessage>`, но это бессмысленно, учитывая возможность полагаться на то, что имитированная функция должна выполняться быстро. Ниже показано, как использовать наш имитированный обработчик:

```
var mocker = new MockHandler (request =>
  new HttpResponseMessage (HttpStatusCode.OK)
  {
    Content = new StringContent ("You asked for " + request.RequestUri)
  });

var client = new HttpClient (mocker);
var response = await client.GetAsync ("http://www.linqpad.net");
string result = await response.Content.ReadAsStringAsync();
Assert.AreEqual ("You asked for http://www.linqpad.net/", result);
```

(`Assert.AreEqual` — метод, который мы ожидаем обнаружить в инфраструктурах модульного тестирования, подобных NUnit.)

### Соединение в цепочки обработчиков с помощью `DelegatingHandler`

За счет создания подклассов класса `DelegatingHandler` можно построить обработчик сообщений, который вызывает другой обработчик (давая в результате цепочку обработчиков). Подобный прием может применяться для реализации специальных протоколов аутентификации, сжатия и шифрования. Ниже приведен код простого обработчика регистрации в журнале:

```
class LoggingHandler : DelegatingHandler
{
  public LoggingHandler (HttpMessageHandler nextHandler)
  {
    InnerHandler = nextHandler;
  }
```

```
  protected async override Task <HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken)
  {
    Console.WriteLine ("Requesting: " + request.RequestUri);
    var response = await base.SendAsync (request, cancellationToken);
    Console.WriteLine ("Got response: " + response.StatusCode);
    return response;
  }
}
```

Обратите внимание, что в переопределенном методе `SendAsync` поддерживается асинхронность. Введение модификатора `async` при переопределении метода, возвращающего задачу, совершенно допустимо, а в данном случае еще и желательно.

В более удачном решении, нежели простой вывод на консоль, можно было бы предложить конструктор, который принимает некоторый вид регистрирующего объекта. А еще лучше было бы принимать пару делегатов `Action<T>`, сообщающих о том, каким образом регистрировать в журнале объекты запроса и ответа.

## Прокси-серверы

*Прокси-сервер* – это посредник, через который могут маршрутизироваться запросы HTTP и FTP. Иногда организации настраивают прокси-сервер как единственное средство, с помощью которого сотрудники могут получать доступ в Интернет – главным образом потому, что это упрощает действия по обеспечению безопасности. Прокси-сервер имеет собственный адрес и может требовать аутентификации, так что доступ в Интернет будет разрешен только избранным пользователям локальной сети.

Объект `WebClient` или `WebRequest` можно проинструктировать на предмет маршрутизации запросов через прокси-сервер посредством объекта `WebProxy`:

```
// Создать объект WebProxy с IP-адресом и портом прокси.
// Дополнительно можно установить
// свойство Credentials, если для прокси требует указания
// имени пользователя/пароля.

WebProxy p = new WebProxy ("192.178.10.49", 808);
p.Credentials = new NetworkCredential ("имя пользователя", "пароль");
// или:
p.Credentials = new NetworkCredential ("имя пользователя", "пароль", "домен");

WebClient wc = new WebClient();
wc.Proxy = p;
  ...

// Та же самая процедура в отношении объекта WebRequest:
WebRequest req = WebRequest.Create ("...");
req.Proxy = p;
```

Чтобы использовать прокси-сервер вместе с `HttpClient`, сначала нужно создать объект `HttpClientHandler`, установить его свойство `Proxy` и затем передать результат конструктору `HttpClient`:

```
WebProxy p = new WebProxy ("192.178.10.49", 808);
p.Credentials = new NetworkCredential ("имя пользователя", "пароль", "домен");

var handler = new HttpClientHandler { Proxy = p };
var client = new HttpClient (handler);
...
```

Если известно, что нет никаких прокси-серверов, то свойство `Proxy` объектов `WebClient` и `WebRequest` полезно установить в `null`. В противном случае инфраструктура .NET Framework может попытаться определить параметры прокси-сервера автоматически, увеличивая длительность запроса на временной промежуток, который может достигать 30 секунд. Если вас беспокоит, почему веб-запросы выполняются медленно, то вполне вероятно, что именно по этой причине.

Класс `HttpClientHandler` имеет также свойство `UseProxy`, которое можно установить в `false` вместо установки в `null` свойства `Proxy` для отмены автоматического определения параметров прокси-сервера.

Если при конструировании объекта `NetworkCredential` указан домен, то будут использоваться протоколы аутентификации на основе Windows. Чтобы задействовать текущего аутентифицированного пользователя Windows, установите статическое свойство `CredentialCache.DefaultNetworkCredentials` в значение свойства `Credentials` прокси-сервера.

В качестве альтернативы частой установке свойства `Proxy` можно установить глобальное стандартное значение:

```
WebRequest.DefaultWebProxy = myWebProxy;
```

или:

```
WebRequest.DefaultWebProxy = null;
```

Такая установка применяется на время существования домена приложения (если только ее не изменит какой-то другой код).

## Аутентификация

Чтобы предоставить сайту HTTP или FTP имя пользователя и пароль, необходимо создать объект `NetworkCredential` и присвоить его свойству `Credentials` экземпляра `WebClient` или `WebRequest`:

```
WebClient wc = new WebClient { Proxy = null };
wc.BaseAddress = "ftp://ftp.albahari.com";

// Провести аутентификацию, после чего выгрузить и загрузить файл на FTP-сервер.
// Тот же самый подход также работает для протоколов HTTP и HTTPS.

string username = "nutshell";
string password = "oreilly";
wc.Credentials = new NetworkCredential (username, password);

wc.DownloadFile ("guestbook.txt", "guestbook.txt");

string data = "Hello from " + Environment.UserName + "!\r\n";
File.AppendAllText ("guestbook.txt", data);

wc.UploadFile ("guestbook.txt", "guestbook.txt");
```

Класс `HttpClient` открывает доступ к тому же самому свойству `Credentials` через `HttpClientHandler`:

```
var handler = new HttpClientHandler();
handler.Credentials = new NetworkCredential (username, password);
var client = new HttpClient (handler);
...
```

Данный прием работает с основанными на диалоговых окнах протоколами аутентификации, такими как Basic и Digest, и расширяется посредством класса `AuthenticationManager`. Он также поддерживает протоколы Windows NTLM и Kerberos (когда при конструировании объекта `NetworkCredential` указано имя домена). Если нужно использовать текущего аутентифицированного пользователя Windows, тогда свойство `Credentials` можно оставить равным `null` и вместо него установить свойство `UseDefaultCredentials` в `true`.

Установка свойства `Credentials` бесполезна при аутентификации на основе форм. Аутентификация на основе форм обсуждается отдельно (в разделе "Аутентификация на основе форм" далее в главе).

Аутентификация в конечном итоге поддерживается подтипом типа `WebRequest` (в данном случае `FtpWebRequest`), который автоматически согласовывает совместимый протокол. Для HTTP может существовать выбор: например, если вы просмотрите начальный ответ от страницы веб-почты сервера Microsoft Exchange, то можете встретить следующие заголовки:

```
HTTP/1.1 401 Unauthorized
Content-Length: 83
Content-Type: text/html
Server: Microsoft-IIS/6.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
WWW-Authenticate: Basic realm="exchange.somedomain.com"
X-Powered-By: ASP.NET
Date: Sat, 05 Aug 2006 12:37:23 GMT
```

Код 401 сигнализирует о том, что авторизация обязательна; заголовки `WWW-Authenticate` указывают, какие будут восприниматься протоколы аутентификации. Однако если сконфигурировать объект `WebClient` или `WebRequest` с корректным именем пользователя и паролем, то это сообщение будет скрыто, поскольку .NET Framework отвечает автоматически, выбирая совместимый протокол аутентификации и затем повторно отправляя исходный запрос с дополнительным заголовком. Например:

```
Authorization: Negotiate TlRMTVNTUAAABAAAt5II2gjACDArAAACAwACACgAAAAQ
ATmKAAAAD0lVDRdPUksHUq9VUA==
```

Такой механизм отличается прозрачностью, но приводит к дополнительному двухстороннему обмену для каждого запроса. Установив свойство `PreAuthenticate` в `true`, дополнительных двухсторонних обменов при последующих запросах к тому же самому URI можно избежать. Данное свойство определено в классе `WebRequest` (и работает только в случае `HttpWebRequest`). Класс `WebClient` не поддерживает указанную возможность в принципе.

### `CredentialCache`

С помощью объекта `CredentialCache` можно принудительно применить конкретный протокол аутентификации. Кеш учетных данных (credential cache) содержит один или большее количество объектов `NetworkCredential`, каждый из которых связан с конкретным протоколом и префиксом URI. Например, во время входа на сервер Exchange Server может возникнуть желание пропустить протокол Basic, т.к. он передает пароли в виде открытого текста:

```
CredentialCache cache = new CredentialCache();
Uri prefix = new Uri ("http://exchange.somedomain.com");
cache.Add (prefix, "Digest",  new NetworkCredential ("joe", "passwd"));
cache.Add (prefix, "Negotiate", new NetworkCredential ("joe", "passwd"));

WebClient wc = new WebClient();
wc.Credentials = cache;
...
```

Протокол аутентификации указывается в форме строки со следующими допустимыми значениями:

```
Basic, Digest, NTLM, Kerberos, Negotiate
```

В этом конкретном примере `WebClient` выберет значение `Negotiate`, потому что сервер не сообщил о поддержке протокола Digest в своих заголовках аутентификации. Здесь Negotiate – это Windows-протокол, который в зависимости от возможностей сервера сводится к Kerberos или NTLM.

Статическое свойство `CredentialCache.DefaultNetworkCredentials` позволяет добавлять текущего аутентифицированного пользователя Windows в кеш учетных данных без необходимости в предоставлении пароля:

```
cache.Add (prefix, "Negotiate", CredentialCache.DefaultNetworkCredentials);
```

### Аутентификация через заголовки с помощью `HttpClient`

В случае использования `HttpClient` есть и другой способ аутентификации, предусматривающий установку заголовка аутентификации напрямую:

```
var client = new HttpClient();
client.DefaultRequestHeaders.Authorization =
  new AuthenticationHeaderValue ("Basic",
    Convert.ToBase64String (Encoding.UTF8.GetBytes ("username:password")));
...
```

Такая стратегия работает и со специальными системами аутентификации вроде OAuth. Заголовки более подробно обсуждаются далее в главе.

## Обработка исключений

В случае ошибки, связанной с сетью или протоколом, классы `WebRequest`, `WebResponse`, `WebClient` и их потоки генерируют исключение `WebException`. Класс `HttpClient` делает то же самое, но затем помещает экземпляр `WebException` внутрь `HttpRequestException`. Конкретную ошибку можно определить с помощью свойства `Status` класса `WebException`, возвращающего тип перечисления `WebExceptionStatus`, который имеет следующие члены:

```
CacheEntryNotFound                RequestCanceled
ConnectFailure                    RequestProhibitedByCachePolicy
ConnectionClosed                  RequestProhibitedByProxy
KeepAliveFailure                  SecureChannelFailure
MessageLengthLimitExceeded        SendFailure
NameResolutionFailure             ServerProtocolViolation
Pending                           Success
PipelineFailure                   Timeout
ProtocolError                     TrustFailure
ProxyNameResolutionFailure        UnknownError
ReceiveFailure
```

Недопустимое доменное имя вызывает ошибку `NameResolutionFailure`, отказ сети — ошибку `ConnectFailure`, а запрос, превысивший тайм-аут длительностью `WebRequest.Timeout` миллисекунд — ошибку `Timeout`.

Ошибки наподобие "Page not found" (страница не найдена), "Moved Permanently" (перемещено постоянно) и "Not Logged In" (не авторизовано) специфичны для протоколов HTTP и FTP, поэтому они все вместе объединены в состояние `ProtocolError`. В классе `HttpClient` упомянутые ошибки не генерируются до тех пор, пока не будет вызван метод `EnsureSuccessStatusCode` на объекте ответа. Прежде чем поступить так, можно получить код состояния, запросив свойство `StatusCode`:

```
var client = new HttpClient();
var response = await client.GetAsync ("http://linqpad.net/foo");
HttpStatusCode responseStatus = response.StatusCode;
```

В случае `WebClient` и `WebRequest/WebResponse` на самом деле потребуется перехватить исключение `WebException` и затем выполнить перечисленные ниже действия.

1. Привести свойство `Response` экземпляра `WebException` к типу `HttpWebResponse` или `FtpWebResponse`.
2. Проверить свойство `Status` объекта ответа (типа перечисления `HttpStatusCode` или `FtpStatusCode`) и/или его свойство `StatusDescription` (строкового типа).

Например:

```
WebClient wc = new WebClient { Proxy = null };
try
{
  string s = wc.DownloadString ("http://www.albahari.com/notthere");
}
catch (WebException ex)
{
  if (ex.Status == WebExceptionStatus.NameResolutionFailure)
    Console.WriteLine ("Bad domain name");  // Недопустимое имя домена
  else if (ex.Status == WebExceptionStatus.ProtocolError)
  {
    HttpWebResponse response = (HttpWebResponse) ex.Response;
    Console.WriteLine (response.StatusDescription);  // "Not Found"
    if (response.StatusCode == HttpStatusCode.NotFound)
      Console.WriteLine ("Not there!");              // "Not there!"
  }
  else throw;
}
```

Если необходим трехзначный код состояния, такой как 401 или 404, просто приведите перечисление `HttpStatusCode` или `FtpStatusCode` к целочисленному типу.

По умолчанию вы никогда не получите ошибку перенаправления, потому что классы `WebClient` и `WebRequest` автоматически следуют ответам перенаправления. Чтобы отключить указанное поведение в объекте `WebRequest`, необходимо установить свойство `AllowAutoRedirect` в `false`.

Ошибками перенаправления являются 301 (Moved Permanently — перемещено постоянно), 302 (Found/Redirect — найдено/перенаправлено) и 307 (Temporary Redirect — перенаправлено временно).

Если исключение сгенерировано из-за некорректного применения классов WebClient и WebRequest, то им вероятнее всего будет InvalidOperationException или ProtocolViolationException, а не WebException.

# Работа с протоколом HTTP

В этом разделе описаны специфичные для HTTP возможности запросов и ответов, поддерживаемые классами WebClient, HttpWebRequest/HttpWebResponse и HttpClient.

## Заголовки

Классы WebClient, WebRequest и HttpClient позволяют добавлять специальные HTTP-заголовки, а также производить перечисление имеющихся заголовков в ответе. Заголовок представляет собой просто пару "ключ/значение" с метаданными, такими как тип содержимого сообщения или программное обеспечение сервера. Ниже показано, как добавить специальный заголовок к запросу и вывести список всех запросов из сообщения ответа внутри объекта WebClient:

```
WebClient wc = new WebClient { Proxy = null };
wc.Headers.Add ("CustomHeader", "JustPlaying/1.0");
wc.DownloadString ("http://www.oreilly.com");

foreach (string name in wc.ResponseHeaders.Keys)
  Console.WriteLine (name + "=" + wc.ResponseHeaders [name]);
```

```
Age=51
X-Cache=HIT from oregano.bp
X-Cache-Lookup=HIT from oregano.bp:3128
Connection=keep-alive
Accept-Ranges=bytes
Content-Length=95433
Content-Type=text/html
...
```

Класс HttpClient взамен открывает доступ к строго типизированным коллекциям со свойствами для стандартных HTTP-заголовков. Свойство DefaultRequestHeaders предназначено для заголовков, которые применяются к каждому запросу:

```
var client = new HttpClient (handler);
client.DefaultRequestHeaders.UserAgent.Add (
  new ProductInfoHeaderValue ("VisualStudio", "2015"));
client.DefaultRequestHeaders.Add ("CustomHeader", "VisualStudio/2015");
```

С другой стороны, свойство Headers класса HttpRequestMessage представляет заголовки, специфичные для запроса.

## Строки запросов

Строка запроса – это просто добавляемая к URI строка со знаком вопроса, которая используется для отправки простых данных серверу. В строке запроса можно указывать множество пар "ключ/значение" с применением следующего синтаксиса:

```
?key1=value1&key2=value2&key3=value3...
```

Класс WebClient предлагает несложный способ добавления строк запросов через свойство в стиле словаря. Следующий код ищет в Google слово "WebClient", отображая страницу результатов на французском языке:

```
WebClient wc = new WebClient { Proxy = null };
wc.QueryString.Add ("q", "WebClient"); // Искать слово WebClient
wc.QueryString.Add ("hl", "fr"); // Отобразить страницу на французском языке
wc.DownloadFile ("http://www.google.com/search", "results.html");
System.Diagnostics.Process.Start ("results.html");
```

Для достижения того же результата с помощью класса `WebRequest` или `HttpClient` потребуется вручную добавить к URI запроса корректно сформатированную строку:

```
string requestURI = "http://www.google.com/search?q=WebClient&hl=fr";
```

Если существует вероятность того, что запрос будет включать нестандартные символы или пробелы, то для создания допустимого URI можно задействовать метод `EscapeDataString` класса `Uri`:

```
string search = Uri.EscapeDataString ("(WebClient OR HttpClient)");
string language = Uri.EscapeDataString ("fr");
string requestURI = "http://www.google.com/search?q=" + search +
                    "&hl=" + language;
```

Результирующий URI выглядит так:

```
http://www.google.com/search?q=(WebClient%20OR%20HttpClient)&hl=fr
```

(Метод `EscapeDataString` похож на `EscapeUriString` за исключением того, что он также кодирует символы вроде `&` и `=`, которые иначе заполонили бы строку запроса.)

Библиотека Microsoft Web Protection (`http://wpl.codeplex.com`) предлагает еще одно решение кодирования/декодирования, при котором принимаются во внимание уязвимости к атакам с помощью межсайтовых сценариев.

## Выгрузка данных формы

Класс `WebClient` предлагает методы `UploadValues` для отправки данных HTML-форме:

```
WebClient wc = new WebClient { Proxy = null };

var data = new System.Collections.Specialized.NameValueCollection();
data.Add ("Name", "Joe Albahari");
data.Add ("Company", "O'Reilly");

byte[] result = wc.UploadValues ("http://www.albahari.com/EchoPost.aspx",
                                 "POST", data);

Console.WriteLine (Encoding.UTF8.GetString (result));
```

Ключи в коллекции `NameValueCollection`, такие как `searchtextbox` и `searchMode`, соответствуют именам полей ввода HTML-формы.

Выгрузка данных формы в большей степени работает через класс `WebRequest`. (Такой подход необходимо выбирать, если должны использоваться средства вроде cookie-наборов.) Соответствующая процедура описана ниже.

1. Установите свойство `ContentType` запроса в `application/x-www-form-urlencoded`, а свойство `Method` запроса — в `POST`.
2. Постройте строку, содержащую данные для выгрузки, которая кодируется следующим образом:

   ```
   имя1=значение1&имя2=значение2&имя3=значение3...
   ```

3. Преобразуйте полученную строку в байтовый массив с помощью метода `Encoding.UTF8.GetBytes`.
4. Установите свойство `ContentLength` веб-запроса в длину результирующего байтового массива.
5. Вызовите метод `GetRequestStream` для веб-запроса и запишите массив данных.
6. Вызовите метод `GetResponse` для чтения ответа от сервера.

Вот как выглядит предыдущий пример, переписанный с применением класса `WebRequest`:

```
var req = WebRequest.Create ("http://www.albahari.com/EchoPost.aspx");
req.Proxy = null;
req.Method = "POST";
req.ContentType = "application/x-www-form-urlencoded";

string reqString = "Name=Joe+Albahari&Company=O'Reilly";
byte[] reqData = Encoding.UTF8.GetBytes (reqString);
req.ContentLength = reqData.Length;

using (Stream reqStream = req.GetRequestStream())
  reqStream.Write (reqData, 0, reqData.Length);

using (WebResponse res = req.GetResponse())
using (Stream resSteam = res.GetResponseStream())
using (StreamReader sr = new StreamReader (resSteam))
  Console.WriteLine (sr.ReadToEnd());
```

В случае класса `HttpClient` взамен создается и наполняется объект `FormUrlEncodedContent`, который затем можно либо передать методу `PostAsync`, либо присвоить свойству `Content` запроса:

```
string uri = "http://www.albahari.com/EchoPost.aspx";
var client = new HttpClient();
var dict = new Dictionary<string,string>
{
  { "Name", "Joe Albahari" },
  { "Company", "O'Reilly" }
};
var values = new FormUrlEncodedContent (dict);
var response = await client.PostAsync (uri, values);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

## Cookie-наборы

Cookie-набор – это строка с парой "имя/значение", которую HTTP-сервер посылает клиенту в заголовке ответа. Клиентский веб-браузер обычно запоминает cookie-наборы и повторяет их для сервера в каждом последующем запросе (к тому же адресу) до тех пор, пока не истечет время их действия. Cookie-набор позволяет серверу знать, что он взаимодействует с тем же самым клиентом, с которым он имел дело минуту назад (или, скажем, вчера), без необходимости в наличии неаккуратной строки запроса в URI.

По умолчанию `HttpWebRequest` игнорирует любые cookie-наборы, полученные от сервера. Чтобы принимать cookie-наборы, следует создать объект `CookieContainer` и присвоить его свойству `CookieContainer` экземпляра `WebRequest`. Затем можно выполнить перечисление cookie-наборов, полученных в ответе:

```
var cc = new CookieContainer();
var request = (HttpWebRequest) WebRequest.Create ("http://www.google.com");
request.Proxy = null;
request.CookieContainer = cc;
using (var response = (HttpWebResponse) request.GetResponse())
{
  foreach (Cookie c in response.Cookies)
  {
    Console.WriteLine (" Name:   " + c.Name);      // Имя
    Console.WriteLine (" Value:  " + c.Value);     // Значение
    Console.WriteLine (" Path:   " + c.Path);      // Путь
    Console.WriteLine (" Domain: " + c.Domain);    // Домен
  }
  // Читать поток ответа...
}
```

```
Name:   PREF
Value:  ID=6b10df1da493a9c4:TM=1179025486:LM=1179025486:S=EJCZri0aWEHlk4tt
Path:   /
Domain: .google.com
```

Чтобы сделать то же самое с помощью класса `HttpClient`, сначала понадобится создать экземпляр `HttpClientHandler`:

```
var cc = new CookieContainer();
var handler = new HttpClientHandler();
handler.CookieContainer = cc;
var client = new HttpClient (handler);
...
```

Фасадный класс `WebClient` cookie-наборы не поддерживает.

Для повторения полученных cookie-наборов в будущих запросах просто присваивайте тот же самый объект `CookieContainer` свойству `CookieContainer` каждого нового экземпляра `WebRequest`, а в случае класса `HttpClient` для выполнения запросов продолжайте использовать тот же самый объект. Объект `CookieContainer` поддерживает возможность сериализации, так что его можно записывать на диск, как объясняется в главе 17. В качестве альтернативы можно начать с нового объекта `CookieContainer`, а затем добавить cookie-наборы вручную:

```
Cookie c = new Cookie ("PREF",
                       "ID=6b10df1da493a9c4:TM=1179...",
                       "/",
                       ".google.com");
freshCookieContainer.Add (c);
```

Третий и четвертый аргументы отражают путь и домен источника. Объект `CookieContainer` на стороне клиента может хранить cookie-наборы из множества разных мест; объект `WebRequest` посылает только те cookie-наборы, путь и домен которых совпадают с путем и доменом сервера.

## Аутентификация на основе форм

В предыдущем разделе было показано, как объект `NetworkCredentials` может удовлетворить требованиям систем аутентификации вроде Basic или NTLM (которые инициируют появление всплывающего диалогового окна в веб-браузере). Однако большинство веб-сайтов требуют аутентификации с применением подхода на основе форм. Введите свое имя пользователя и пароль в текстовые поля, которые являются

частью HTML-формы, декорированной соответствующей корпоративной графикой, щелкните на кнопке для отправки данных и затем получите cookie-набор после успешной аутентификации. Этот cookie-набор предоставит более высокие полномочия при просмотре страниц на веб-сайте. Посредством `WebRequest` или `HttpClient` все действия можно делать программно, имея доступ к возможностям, которые обсуждались в предшествующих двух разделах. Такой прием может быть удобен для тестирования или для автоматизации в ситуациях, когда подходящий API-интерфейс отсутствует.

Типичный веб-сайт, реализующий аутентификацию с помощью форм, будет содержать примерно такую HTML-разметку:

```
<form action="http://www.somesite.com/login" method="post">
  <input type="text" id="user" name="username">
  <input type="password" id="pass" name="password">
  <button type="submit" id="login-btn">Log In</button>
</form>
```

Вот как войти на данный веб-сайт с помощью классов `WebRequest/WebResponse`:

```
string loginUri = "http://www.somesite.com/login";
string username = "username";   // (Имя пользователя)
string password = "password";   // (Пароль)
string reqString = "username=" + username + "&password=" + password;
byte[] requestData = Encoding.UTF8.GetBytes (reqString);

CookieContainer cc = new CookieContainer();
var request = (HttpWebRequest)WebRequest.Create (loginUri);
request.Proxy = null;
request.CookieContainer = cc;
request.Method = "POST";

request.ContentType = "application/x-www-form-urlencoded";
request.ContentLength = requestData.Length;

using (Stream s = request.GetRequestStream())
  s.Write (requestData, 0, requestData.Length);

using (var response = (HttpWebResponse) request.GetResponse())
  foreach (Cookie c in response.Cookies)
    Console.WriteLine (c.Name + " = " + c.Value);

// Теперь мы вошли на сайт. До тех пор, пока мы будем указывать cc последующим
// объектам WebRequest, мы будем считаться аутентифицированным пользователем.
```

А вот как сделать то же самое посредством `HttpClient`:

```
string loginUri = "http://www.somesite.com/login";
string username = "username";
string password = "password";

CookieContainer cc = new CookieContainer();
var handler = new HttpClientHandler { CookieContainer = cc };
var request = new HttpRequestMessage (HttpMethod.Post, loginUri);
request.Content = new FormUrlEncodedContent (new Dictionary<string,string>
{
  { "username", username },
  { "password", password }
});
var client = new HttpClient (handler);
var response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
...
```

## SSL

Классы WebClient, HttpClient и WebRequest используют протокол SSL автоматически, когда указывается префикс https:. Единственная сложность, которая может возникнуть, связана с недействительными сертификатами X.509. Если сертификат сайта сервера не является допустимым в каком-то аспекте (например, если сертификат тестовый), тогда при попытке взаимодействия генерируется исключение. Чтобы обойти это, к статическому классу ServicePointManager можно присоединить специальное средство проверки достоверности сертификата:

```
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;
...
static void ConfigureSSL()
{
  ServicePointManager.ServerCertificateValidationCallback = CertChecker;
}
```

Свойство ServerCertificateValidationCallback представляет собой делегат. Если он возвращает true, то сертификат принимается:

```
static bool CertChecker (object sender, X509Certificate certificate,
                         X509Chain chain, SslPolicyErrors errors)
{
  // Возвратить true, если сертификат подходит
  ...
}
```

# Реализация HTTP-сервера

С помощью класса HttpListener можно реализовать собственный HTTP-сервер .NET. Ниже приведен код простого сервера, который прослушивает порт 51111, ожидает одиночный клиентский запрос и возвращает однострочный ответ.

```
static void Main()
{
  ListenAsync();                    // Запустить сервер.
  WebClient wc = new WebClient();   // Построить клиентский запрос.
  Console.WriteLine (wc.DownloadString
    ("http://localhost:51111/MyApp/Request.txt"));
}

async static void ListenAsync()
{
  HttpListener listener = new HttpListener();
  listener.Prefixes.Add ("http://localhost:51111/MyApp/");  // Прослушивать
  listener.Start();                                         // порт 51111.

  // Ожидать поступления клиентского запроса:
  HttpListenerContext context = await listener.GetContextAsync();

  // Ответить на запрос:
  string msg = "You asked for: " + context.Request.RawUrl;
  context.Response.ContentLength64 = Encoding.UTF8.GetByteCount (msg);
  context.Response.StatusCode = (int) HttpStatusCode.OK;

  using (Stream s = context.Response.OutputStream)
```

```
    using (StreamWriter writer = new StreamWriter (s))
      await writer.WriteAsync (msg);

    listener.Stop();
  }
ВЫВОД: You asked for: /MyApp/Request.txt
```

Внутренне класс `HttpListener` не применяет .NET-объекты `Socket`, а обращается к API-интерфейсу HTTP-серверов Windows (Windows HTTP Server API). Это позволяет множеству приложений на компьютере прослушивать один и тот же IP-адрес и порт – при условии, что каждое из них регистрирует отличающиеся адресные префиксы. В показанном выше примере мы регистрируем префикс `http://localhost/myapp`, поэтому другое приложение способно прослушивать тот же самый IP-адрес и порт с другим префиксом, таким как `http://localhost/anotherapp`. Это имеет значение, потому что открытие новых портов в корпоративных брандмауэрах может быть затруднено из-за политик, действующих внутри компании.

Объект `HttpListener` ожидает следующего клиентского запроса, когда вызывается метод `GetContext`, возвращая объект со свойствами `Request` и `Response`. Оба они аналогичны объектам `WebRequest` и `WebResponse`, но со стороны сервера. Например, можно читать и записывать заголовки и cookie-наборы для объектов запросов и ответов почти так же, как это делается на стороне клиента.

Основываясь на ожидаемой клиентской аудитории, можно выбрать полноту, с которой будут поддерживаться функциональные возможности протокола HTTP. В качестве самого минимума для каждого запроса должны устанавливаться длина содержимого и код состояния.

Ниже представлен код простого сервера веб-страниц, реализованного *асинхронным образом*:

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading.Tasks;

class WebServer
{
  HttpListener _listener;
  string _baseFolder;  // Папка для веб-страниц.

  public WebServer (string uriPrefix, string baseFolder)
  {
    _listener = new HttpListener();
    _listener.Prefixes.Add (uriPrefix);
    _baseFolder = baseFolder;
  }

  public async void Start()
  {
    _listener.Start();
    while (true)
      try
      {
        var context = await _listener.GetContextAsync();
        Task.Run (() => ProcessRequestAsync (context));
      }
      catch (HttpListenerException) { break; }  // Прослушиватель остановлен.
```

```
    catch (InvalidOperationException) { break; } // Прослушиватель остановлен.
  }
  public void Stop() { _listener.Stop(); }
  async void ProcessRequestAsync (HttpListenerContext context)
  {
    try
    {
      string filename = Path.GetFileName (context.Request.RawUrl);
      string path = Path.Combine (_baseFolder, filename);
      byte[] msg;
      if (!File.Exists (path))
      {
        Console.WriteLine ("Resource not found: " + path); // Ресурс не найден
        context.Response.StatusCode = (int) HttpStatusCode.NotFound;
        msg = Encoding.UTF8.GetBytes ("Sorry, that page does not exist");
      }
      else
      {
        context.Response.StatusCode = (int) HttpStatusCode.OK;
        msg = File.ReadAllBytes (path);
      }
      context.Response.ContentLength64 = msg.Length;
      using (Stream s = context.Response.OutputStream)
        await s.WriteAsync (msg, 0, msg.Length);
    }
    catch (Exception ex) { Console.WriteLine ("Request error: " + ex); }
                                                  // Ошибка запроса
  }
}
```

А вот метод `Main`, который приводит все в действие:

```
static void Main()
{
  // Прослушивать порт 51111, обслуживая файлы в d:\webroot:
  var server = new WebServer ("http://localhost:51111/", @"d:\webroot");
  try
  {
    server.Start();
    Console.WriteLine ("Server running... press Enter to stop");
                  // Сервер выполняется... для его останова нажмите <Enter>
    Console.ReadLine();
  }
  finally { server.Stop(); }
}
```

Представленный код можно протестировать на стороне клиента с помощью любого браузера; URI в данном случае будет выглядеть как `http://localhost:51111/` плюс имя веб-страницы.

Прослушиватель `HttpListener` не запустится, когда за тот же самый порт соперничает другое программное обеспечение (если только оно также не использует Windows HTTP Server API). Примеры приложений, которые могут прослушивать стандартный порт 80, включают веб-сервер и программы для одноранговых сетей, подобные Skype.

Применение асинхронных функций делает наш сервер масштабируемым и эффективным. Однако его запуск в потоке пользовательского интерфейса будет препятствовать масштабируемости, т.к. для каждого *запроса* управление должно возвращаться в поток пользовательского интерфейса после каждого `await`. Привнесение таких накладных расходов не имеет смысла, особенно с учетом того, что разделяемое состояние отсутствует, поэтому в сценарии с пользовательским интерфейсом мы могли бы поступить следующим образом:

```
Task.Run (Start);
```

или же вызвать `ConfigureAwait(false)` после вызова `GetContextAsync`.

Обратите внимание, что для вызова метода `ProcessRequestAsync` мы используем `Task.Run` несмотря на то, что упомянутый метод уже является асинхронным. Это позволяет вызывающему коду обработать другой запрос *немедленно* вместо того, чтобы сначала ожидать завершения синхронной фазы метода (вплоть до первого `await`).

# Использование FTP

Для выполнения простых операций выгрузки и загрузки FTP можно применять класс `WebClient`, как делалось раньше:

```
WebClient wc = new WebClient { Proxy = null };
wc.Credentials = new NetworkCredential ("nutshell", "oreilly");
wc.BaseAddress = "ftp://ftp.albahari.com";
wc.UploadString ("tempfile.txt", "hello!");
Console.WriteLine (wc.DownloadString ("tempfile.txt"));  // hello!
```

Тем не менее, использование FTP не ограничено лишь выгрузкой и загрузкой файлов. Протокол FTP также поддерживает набор команд, или "методов", определенных как строковые константы в классе `WebRequestMethods.Ftp`:

```
AppendFile
DeleteFile
DownloadFile
GetDateTimestamp
GetFileSize
ListDirectory
ListDirectoryDetails
MakeDirectory
PrintWorkingDirectory
RemoveDirectory
Rename
UploadFile
UploadFileWithUniqueName
```

Чтобы запустить одну из перечисленных команд, необходимо присвоить ее строковую константу свойству `Method` веб-запроса, после чего вызвать метод `GetResponse`. Ниже показано, как получить список файлов в каталоге:

```
var req = (FtpWebRequest) WebRequest.Create ("ftp://ftp.albahari.com");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");
req.Method = WebRequestMethods.Ftp.ListDirectory;

using (WebResponse resp = req.GetResponse())
using (StreamReader reader = new StreamReader (resp.GetResponseStream()))
  Console.WriteLine (reader.ReadToEnd());
```

```
РЕЗУЛЬТАТ:
.
..
guestbook.txt
tempfile.txt
test.doc
```

В случае получения списка файлов в каталоге для извлечения результата необходимо читать поток ответа. Тем не менее, большинство других команд не требуют данного шага. Например, чтобы получить результат команды `GetFileSize`, нужно просто обратиться к свойству `ContentLength` объекта ответа:

```
var req = (FtpWebRequest) WebRequest.Create (
                          "ftp://ftp.albahari.com/tempfile.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");

req.Method = WebRequestMethods.Ftp.GetFileSize;

using (WebResponse resp = req.GetResponse())
  Console.WriteLine (resp.ContentLength);  // 6
```

Команда `GetDateTimestamp` работает в похожей манере, но только запрашивается свойство `LastModified` объекта ответа. Это требует приведения к типу `FtpWebResponse`:

```
...
req.Method = WebRequestMethods.Ftp.GetDateTimestamp;

using (var resp = (FtpWebResponse) req.GetResponse() )
  Console.WriteLine (resp.LastModified);
```

Для применения команды `Rename` придется указать в свойстве `RenameTo` объекта запроса новое имя файла (без префикса в виде каталога). Например, ниже показано, как переименовать файл `tempfile.txt` в `deleteme.txt` в каталоге `incoming`:

```
var req = (FtpWebRequest) WebRequest.Create (
                          "ftp://ftp.albahari.com/tempfile.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");

req.Method = WebRequestMethods.Ftp.Rename;
req.RenameTo = "deleteme.txt";

req.GetResponse().Close();  // Выполнить переименование
```

А вот так файл можно удалить:

```
var req = (FtpWebRequest) WebRequest.Create (
                          "ftp://ftp.albahari.com/deleteme.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");

req.Method = WebRequestMethods.Ftp.DeleteFile;

req.GetResponse().Close();  // Выполнить удаление
```

Во всех рассмотренных примерах обычно должен использоваться блок обработки исключений, предназначенный для перехвата ошибок сети и протокола. Типичный блок `catch` выглядит следующим образом:

```
catch (WebException ex)
{
```

```
    if (ex.Status == WebExceptionStatus.ProtocolError)
    {
      // Получить дополнительные детали, связанные с ошибкой:
      var response = (FtpWebResponse) ex.Response;
      FtpStatusCode errorCode = response.StatusCode;
      string errorMessage = response.StatusDescription;
      ...
    }
    ...
  }
```

# Использование DNS

Статический класс `Dns` инкапсулирует службу доменных имен (Domain Name Service), которая осуществляет преобразования между низкоуровневыми IP-адресами наподобие 66.135.192.87 и понятными для человека доменными именами, такими как `ebay.com`.

Метод `GetHostAddresses` преобразует доменное имя в IP-адрес (или адреса):

```
foreach (IPAddress a in Dns.GetHostAddresses ("albahari.com"))
  Console.WriteLine (a.ToString());  // 205.210.42.167
```

Метод `GetHostEntry` двигается в обратном направлении, преобразуя IP-адрес в доменное имя:

```
IPHostEntry entry = Dns.GetHostEntry ("205.210.42.167");
Console.WriteLine (entry.HostName);  // albahari.com
```

Метод `GetHostEntry` также принимает объект `IPAddress`, поэтому IP-адрес можно указывать в виде байтового массива:

```
IPAddress address = new IPAddress (new byte[] { 205, 210, 42, 167 });
IPHostEntry entry = Dns.GetHostEntry (address);
Console.WriteLine (entry.HostName);  // albahari.com
```

В случае применения класса вроде `WebRequest` или `TcpClient` доменные имена преобразуются в IP-адреса автоматически. Однако если в приложении планируется выполнять множество сетевых запросов к одному и тому же адресу, то производительность иногда можно увеличить, сначала используя класс `Dns` для явного преобразования доменного имени в IP-адрес и затем организовав с ним коммуникации напрямую. Это позволяет избежать повторяющихся циклов двухстороннего обмена для преобразования того же самого доменного имени и может быть полезно при работе с транспортным уровнем (через класс `TcpClient`, `UdpClient` или `Socket`).

Класс `Dns` также предлагает асинхронные методы, которые возвращают объекты задач, допускающих ожидание:

```
foreach (IPAddress a in await Dns.GetHostAddressesAsync ("albahari.com"))
  Console.WriteLine (a.ToString());
```

# Отправка сообщений электронной почты с помощью `SmtpClient`

Класс `SmtpClient` из пространства имен `System.Net.Mail` позволяет отправлять сообщения электронной почты с применением простого протокола передачи почты (Simple Mail Transfer Protocol – SMTP). Чтобы отправить обычное текстовое сообще-

ние, необходимо создать экземпляр SmtpClient, указать в его свойстве Host адрес SMTP-сервера и затем вызвать метод Send:

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myisp.net";
client.Send ("from@adomain.com", "to@adomain.com", "subject", "body");
```

С целью противостояния спамерам большинство SMTP-серверов в Интернете принимают подключения только от абонентов поставщиков услуг Интернета, поэтому вам понадобится адрес SMTP, который соответствует текущему подключению.

При конструировании объекта MailMessage доступны и другие варианты, включая возможность добавления вложений:

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myisp.net";
MailMessage mm = new MailMessage();

mm.Sender = new MailAddress ("kay@domain.com", "Kay");
mm.From   = new MailAddress ("kay@domain.com", "Kay");
mm.To.Add  (new MailAddress ("bob@domain.com", "Bob"));
mm.CC.Add  (new MailAddress ("dan@domain.com", "Dan"));
mm.Subject = "Hello!";
mm.Body = "Hi there. Here's the photo!";
mm.IsBodyHtml = false;
mm.Priority = MailPriority.High;

Attachment a = new Attachment ("photo.jpg",
                               System.Net.Mime.MediaTypeNames.Image.Jpeg);
mm.Attachments.Add (a);
client.Send (mm);
```

Класс SmtpClient позволяет указывать в свойстве Credentials учетные данные для серверов, требующих аутентификации, устанавливать свойство EnableSsl в true, если поддерживается протокол SSL, а также задавать в свойстве Port нестандартный порт TCP. Изменяя свойство DeliveryMethod, можно заставить SmtpClient использовать сервер IIS для отправки почтовых сообщений или просто записывать каждое сообщение в файл .eml внутри указанного каталога:

```
SmtpClient client = new SmtpClient();
client.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
client.PickupDirectoryLocation = @"c:\mail";
```

# Использование TCP

TCP и UDP являются протоколами транспортного уровня, на основе которых построено большинство служб Интернета и локальных вычислительных сетей. Протоколы HTTP, FTP и SMTP работают с TCP, а DNS – с UDP. Протокол TCP ориентирован на подключение и поддерживает механизмы обеспечения надежности; UDP является протоколом без установления подключения, характеризуется низкими накладными расходами и поддерживает широковещательную передачу. Протокол *BitTorrent* применяет UDP, как это делает Voice over IP (VoIP).

Транспортный уровень предлагает более высокую гибкость – и потенциально лучшую производительность – по сравнению с более высокими уровнями, но требует самостоятельного решения таких задач, как аутентификация и шифрование.

Благодаря поддержке TCP в .NET можно работать либо с легкими в пользовании фасадными классами TcpClient и TcpListener, либо с обладающим обширными

возможностями классом Socket. (В действительности их можно сочетать, поскольку класс TcpClient через свое свойство Client открывает доступ к внутреннему объекту Socket.) Класс Socket предоставляет больше вариантов конфигурации и разрешает прямой доступ к сетевому уровню (IP) и протоколам, не основанным на Интернете, таким как SPX/IPX от Novell.

(Коммуникации TCP и UDP также возможны через типы WinRT – см. раздел "TCP в Windows Runtime" далее в главе.)

Как и другие протоколы, TCP различает концепции клиента и сервера: клиент инициирует запрос, а сервер запрос ожидает. Ниже представлена базовая структура для синхронного запроса клиента TCP:

```
using (TcpClient client = new TcpClient())
{
  client.Connect ("address", port);
  using (NetworkStream n = client.GetStream())
  {
    // Выполнять чтение и запись в сетевой поток...
  }
}
```

Метод Connect класса TcpClient блокируется вплоть до установления подключения (его асинхронным эквивалентом является метод ConnectAsync). Затем объект NetworkStream предоставляет средство двухсторонних коммуникаций для передачи и получения байтов данных из сервера.

Простой сервер TCP выглядит следующим образом:

```
TcpListener listener = new TcpListener (<IP-адрес>, port);
listener.Start();

while (keepProcessingRequests)
  using (TcpClient c = listener.AcceptTcpClient())
  using (NetworkStream n = c.GetStream())
  {
    // Выполнять чтение и запись в сетевой поток...
  }

listener.Stop();
```

Объект TcpListener требует указания локального IP-адреса для прослушивания (например, компьютер с двумя сетевыми адаптерами может иметь два адреса). Чтобы обеспечить прослушивание всех локальных IP-адресов (или только одного из них), можно применять поле IPAddress.Any. Вызов метода AcceptTcpClient класса TcpListener блокируется до тех пор, пока не будет получен клиентский запрос (есть также асинхронная версия этого метода), после чего мы вызываем метод GetStream, в точности как поступали на стороне клиента.

При работе на транспортном уровне необходимо принять решение относительно протокола о том, кто и когда передает данные – почти как при использовании портативной рации. Если оба участника начинают говорить или слушать одновременно, то связь будет нарушена.

Давайте создадим протокол, при котором клиент начинает общение первым, сказав "Hello", после чего сервер отвечает фразой "Hello right back!". Ниже показан соответствующий код:

```
using System;
using System.IO;
using System.Net;
```

```
using System.Net.Sockets;
using System.Threading;

class TcpDemo
{
  static void Main()
  {
    new Thread (Server).Start();  // Запустить серверный метод параллельно.
    Thread.Sleep (500);           // Предоставить серверу время для запуска.
    Client();
  }
  static void Client()
  {
    using (TcpClient client = new TcpClient ("localhost", 51111))
    using (NetworkStream n = client.GetStream())
    {
      BinaryWriter w = new BinaryWriter (n);
      w.Write ("Hello");
      w.Flush();
      Console.WriteLine (new BinaryReader (n).ReadString());
    }
  }
  static void Server() // Обрабатывает одиночный клиентский запрос и завершается.
  {
    TcpListener listener = new TcpListener (IPAddress.Any, 51111);
    listener.Start();
    using (TcpClient c = listener.AcceptTcpClient())
    using (NetworkStream n = c.GetStream())
    {
      string msg = new BinaryReader (n).ReadString();
      BinaryWriter w = new BinaryWriter (n);
      w.Write (msg + " right back!");
      w.Flush();  // Должен быть вызван метод Flush, потому
    }             // что мы не освобождаем средство записи.
    listener.Stop();
  }
}

// ВЫВОД: Hello right back!
```

В данном примере мы применяем закольцовывание `localhost`, чтобы запустить клиент и сервер на одной машине. Мы произвольно выбрали порт из свободного диапазона (с номером больше 49152) и использовали классы `BinaryWriter` и `BinaryReader` для кодирования текстовых сообщений. Мы не закрывали и не освобождали средства чтения и записи, чтобы сохранить поток `NetworkStream` в открытом состоянии вплоть до завершения взаимодействия.

Классы `BinaryReader` и `BinaryWriter` могут показаться странным выбором для чтения и записи строк. Тем не менее, по сравнению с классами `StreamReader` и `StreamWriter` эти классы обладают важным преимуществом: они дополняют строки целочисленными префиксами, указывающими длину, так что `BinaryReader` всегда знает, сколько байтов должно быть прочитано. Если вызвать метод `StreamReader.ReadToEnd`, то может возникнуть блокировка на неопределенное время, т.к. `NetworkStream` не имеет признака окончания. После того, как подключение открыто, сетевой поток никогда не может быть уверен в том, что клиент не собирается передавать дополнительную порцию данных.

В действительности класс `StreamReader` полностью запрещено применять вместе с `NetworkStream`, даже если вы планируете вызывать только метод `ReadLine`. Причина в том, что класс `StreamReader` имеет буфер опережающего чтения, который может привести к чтению большего числа байтов, чем доступно в текущий момент, и бесконечному блокированию (или до возникновения тайм-аута сокета). Другие потоки, такие как `FileStream`, не страдают подобной несовместимостью с классом `StreamReader`, потому что они поддерживают определенный признак *окончания*, при достижении которого метод `Read` немедленно завершается, возвращая значение 0.

## Параллелизм и TCP

Классы `TcpClient` и `TcpListener` предлагают асинхронные методы на основе задач для реализации масштабируемого параллелизма. Их использование сводится просто к замене вызовов блокирующих методов версиями `*Async` этих методов и применению `await` к возвращаемым ими объектам задач.

В следующем примере мы создадим асинхронный сервер TCP, который принимает запросы длиной 5000 байтов, меняет порядок следования байтов на противоположный и затем отправляет их обратно клиенту:

```
async void RunServerAsync ()
{
  var listener = new TcpListener (IPAddress.Any, 51111);
  listener.Start ();
  try
  {
    while (true)
      Accept (await listener.AcceptTcpClientAsync ());
  }
  finally { listener.Stop(); }
}

async Task Accept (TcpClient client)
{
  await Task.Yield ();
  try
  {
    using (client)
    using (NetworkStream n = client.GetStream ())
    {
      byte[] data = new byte [5000];

      int bytesRead = 0; int chunkSize = 1;
      while (bytesRead < data.Length && chunkSize > 0)
        bytesRead += chunkSize =
          await n.ReadAsync (data, bytesRead, data.Length - bytesRead);

      Array.Reverse (data);     // Поменять порядок следования байтов
                                // на противоположный.
      await n.WriteAsync (data, 0, data.Length);
    }
  }
  catch (Exception ex) { Console.WriteLine (ex.Message); }
}
```

Программа масштабируема в том, что она не блокирует поток на время выполнения запроса. Таким образом, если 1000 клиентов одновременно подключаются к сети с использованием медленных каналов (так что от начала до конца каждого запроса проходит, скажем, несколько секунд), то программа не потребует на это время 1000 потоков (в отличие от синхронного решения). Взамен она арендует потоки только на краткие периоды времени, чтобы выполнить код до и после выражений `await`.

# Получение почты POP3 с помощью TCP

Поддержка на прикладном уровне протокола POP3 в .NET Framework отсутствует, поэтому для получения почты из сервера POP3 придется работать на уровне TCP. К счастью, данный протокол прост; взаимодействие в POP3 выглядит следующим образом:

| Клиент | Почтовый сервер | Примечания |
|---|---|---|
| Клиент подключается... | `+OK Hello there.` | Приветственное сообщение |
| `USER joe` | `+OK Password required.` | |
| `PASS password` | `+OK Logged in.` | |
| `LIST` | `+OK`<br>`1 1876`<br>`2 5412`<br>`3 845`<br>`.` | Перечисляет идентификаторы и размеры файлов для каждого сообщения на сервере |
| `RETR 1` | `+OK 1876 octets`<br>*Содержимое сообщения #1...*<br>`.` | Извлекает сообщение с указанным идентификатором |
| `DELE 1` | `+OK Deleted.` | Удаляет сообщение из сервера |
| `QUIT` | `+OK Bye-bye.` | |

Каждая команда и ответ завершаются символом новой строки (<CR>+<LF>) за исключением многострочных команд `LIST` и `RETR`, которые завершаются одиночной точкой в отдельной строке. Поскольку мы не можем применять класс `StreamReader` с `NetworkStream`, начнем с написания вспомогательного метода, предназначенного для чтения строки текста без буферизации:

```
static string ReadLine (Stream s)
{
  List<byte> lineBuffer = new List<byte>();
  while (true)
  {
    int b = s.ReadByte();
    if (b == 10 || b < 0) break;
    if (b != 13) lineBuffer.Add ((byte)b);
  }
  return Encoding.UTF8.GetString (lineBuffer.ToArray());
}
```

Нам еще понадобится вспомогательный метод для отправки команды. Так как мы всегда ожидаем получения ответа, начинающегося с +OK, читать и проверять ответ можно в одно и то же время:

```
static void SendCommand (Stream stream, string line)
{
  byte[] data = Encoding.UTF8.GetBytes (line + "\r\n");
  stream.Write (data, 0, data.Length);
  string response = ReadLine (stream);
  if (!response.StartsWith ("+OK"))
    throw new Exception ("POP Error: " + response);  // Ошибка протокола POP
}
```

При наличии таких методов решить задачу извлечения почты довольно легко. Мы устанавливаем подключение TCP на порте 110 (стандартный порт POP3) и затем начинаем взаимодействие с сервером. В этом примере мы записываем каждое почтовое сообщение в произвольно именованный файл с расширением .eml и удаляем сообщение из сервера:

```
using (TcpClient client = new TcpClient ("mail.isp.com", 110))
using (NetworkStream n = client.GetStream())
{
  ReadLine (n);                        // Прочитать приветственное сообщение.
  SendCommand (n, "USER username");
  SendCommand (n, "PASS password");
  SendCommand (n, "LIST");             // Извлечь идентификаторы сообщений.
  List<int> messageIDs = new List<int>();
  while (true)
  {
    string line = ReadLine (n);        // Например, "1 1876"
    if (line == ".") break;
    messageIDs.Add (int.Parse (line.Split (' ')[0] ));  // Идентификатор
                                                        // сообщения.
  }
  foreach (int id in messageIDs)       // Извлечь каждое сообщение.
  {
    SendCommand (n, "RETR " + id);
    string randomFile = Guid.NewGuid().ToString() + ".eml";
    using (StreamWriter writer = File.CreateText (randomFile))
      while (true)
      {
        string line = ReadLine (n);    // Прочитать следующую строку сообщения.
        if (line == ".") break;        // Одиночная точка - конец сообщения.
        if (line == "..") line = "."; // Избавиться от двойной точки.
        writer.WriteLine (line);       // Записать в выходной файл.
      }
    SendCommand (n, "DELE " + id);     // Удалить сообщение из сервера.
  }
  SendCommand (n, "QUIT");
}
```

# TCP в Windows Runtime

В Windows Runtime также доступна функциональность TCP через пространство имен `Windows.Networking.Sockets`. Как и в реализации .NET, есть два основных класса, которые обеспечивают поддержку ролей сервера и клиента. В WinRT эти классы называются `StreamSocketListener` и `StreamSocket`.

Показанный ниже метод запускает сервер на порте 51111 и ожидает подключения клиента. Затем он читает одиночное сообщение, содержащее строку с префиксом-длиной:

```
async void Server()
{
  var listener = new StreamSocketListener();
  listener.ConnectionReceived += async (sender, args) =>
  {
    using (StreamSocket socket = args.Socket)
    {
      var reader = new DataReader (socket.InputStream);
      await reader.LoadAsync (4);
      uint length = reader.ReadUInt32();
      await reader.LoadAsync (length);
      Debug.WriteLine (reader.ReadString (length));
    }
    listener.Dispose(); // Закрыть прослушиватель после одного сообщения.
  };
  await listener.BindServiceNameAsync ("51111");
}
```

В приведенном примере вместо преобразования в .NET-объект `Stream` и последующего использования класса `BinaryReader` для чтения из входного потока применяется тип WinRT по имени `DataReader` (из пространства имен `Windows.Networking`). Класс `DataReader` довольно похож на `BinaryReader` за исключением того, что он поддерживает асинхронность. Метод `LoadAsync` асинхронно читает указанное количество байтов во внутренний буфер, который затем позволяет вызывать такие методы, как `ReadUInt32` или `ReadString`. Идея в том, что если нужно, предположим, прочитать 1000 целочисленных значений в строке, то сначала следует вызвать метод `LoadAsync` со значением 4000, а затем в цикле 1000 раз вызвать метод `ReadInt32`. Это позволит избежать накладных расходов, связанных с вызовом асинхронных операций в цикле (т.к. каждая асинхронная операция сопровождается небольшими накладными расходами).

Классы `DataReader/DataWriter` имеют свойство `ByteOrder`, предназначенное для управления форматом кодирования чисел — "старший байт перед младшим" или "старший байт после младшего". По умолчанию принят формат "старший байт перед младшим".

Объект `StreamSocket`, получаемый из ожидаемого метода `AcceptAsync`, имеет отдельные входной и выходной потоки. Таким образом, чтобы записать сообщение обратно, мы должны использовать поток `OutputStream` сокета. Проиллюстрировать применение `OutputStream` и `DataWriter` можно посредством соответствующего кода клиента:

```
async void Client()
{
  using (var socket = new StreamSocket())
  {
    await socket.ConnectAsync (new HostName ("localhost"), "51111",
                               SocketProtectionLevel.PlainSocket);
    var writer = new DataWriter (socket.OutputStream);
    string message = "Hello!";
    uint length = (uint) Encoding.UTF8.GetByteCount (message);
    writer.WriteUInt32 (length);
    writer.WriteString (message);
    await writer.StoreAsync();
  }
}
```

Мы начинаем с создания объекта `StreamSocket` напрямую, после чего вызываем метод `ConnectAsync` с указанием имени хоста и номера порта. (Конструктору класса `HostName` можно передавать либо имя DNS, либо строку с IP-адресом.) За счет указания `SocketProtectionLevel.Ssl` можно затребовать шифрование SSL (если оно сконфигурировано на сервере).

Мы снова вместо класса `BinaryWriter` из .NET используем класс `DataWriter` из WinRT и записываем длину строки (измеренную в байтах, но не в символах), а за ней саму строку, закодированную с помощью UTF-8. Наконец, мы вызываем метод `StoreAsync`, который записывает буфер в поддерживающий поток, и закрываем сокет.

# 17

# Сериализация

Настоящая глава посвящена сериализации и десериализации – механизмам, с помощью которых объекты могут быть представлены в простой текстовой или двоичной форме. Если не указано особо, то все типы, упоминаемые в главе, находятся в следующих пространствах имен:

```
System.Runtime.Serialization
System.Xml.Serialization
```

## Концепции сериализации

*Сериализация* – это действие по превращению находящегося в памяти объекта или *графа объектов* (набора объектов, ссылающихся друг на друга) в плоское представление в виде потока байтов или XML-узлов, который можно сохранять или передавать. *Десериализация* работает в противоположном направлении, получая поток данных и восстанавливая его в объект или граф объектов в памяти. Сериализация и десериализация обычно используются для решения следующих задач:

- передача объектов по сети или за границы приложения;
- сохранение представлений объектов внутри файла или базы данных.

Еще одно менее распространенное применение связано с глубоким копированием объектов. Механизмы сериализации на основе контрактов данных и XML могут также использоваться в качестве универсальных инструментов для загрузки и сохранения XML-файлов с известной структурой.

Инфраструктура .NET Framework поддерживает сериализацию и десериализацию как с точки зрения клиентов, желающих сериализировать и десериализировать объекты, так и с точки зрения типов, которым требуется определенный контроль над тем, как они сериализируются.

### Механизмы сериализации

В .NET Framework доступны четыре техники выполнения сериализации:

- сериализатор на основе контрактов данных;
- двоичный сериализатор (в настольных приложениях);
- сериализатор (основанный на атрибутах) XML (`XmlSerializer`);
- интерфейс `IXmlSerializable`.

Первые три из них представляют собой "механизмы" сериализации, которые делают большую часть или даже всю работу самостоятельно. Последняя техника является просто приемом выполнения сериализации с применением классов `XmlReader` и `XmlWriter`. Интерфейс `IXmlSerializable` может работать в сочетании с сериализатором на основе контрактов данных или с `XmlSerializer`, обеспечивая решение более сложных задач сериализации на основе XML.

В табл. 17.1 приведены сравнительные оценки всех механизмов. Чем больше указано звездочек, тем выше (и лучше) оценка.

**Таблица 17.1. Сравнение механизмов сериализации**

| Функциональная возможность | Сериализатор контрактов данных | Двоичный сериализатор | `XmlSerializer` | `IXmlSerializable` |
|---|---|---|---|---|
| Уровень автоматизации | *** | ***** | **** | * |
| Привязка к типам | По выбору | Тесная | Слабая | Слабая |
| Переносимость версий | ***** | *** | ***** | ***** |
| Предохранение объектных ссылок | По выбору | Да | Нет | По выбору |
| Возможность сериализации неоткрытых полей | Да | Да | Нет | Да |
| Пригодность к обмену сообщениями с возможностью взаимодействия | ***** | ** | **** | **** |
| Гибкость в чтении/записи XML-файлов | ** | – | **** | ***** |
| Сжатие вывода | ** | **** | ** | ** |
| Производительность | *** | **** | От * до *** | *** |

Оценки для `IXmlSerializable` предполагают, что вы вручную написали оптимальный код, используя классы `XmlReader` и `XmlWriter`. Для достижения хорошей производительности механизм сериализации XML требует повторного применения того же самого объекта `XmlSerializer`.

### Почему механизмов три?

Причина наличия трех механизмов сериализации отчасти историческая. Изначально перед сериализацией в .NET Framework стояли две разные цели:

- сериализация графов объектов .NET с обеспечением точности типов и ссылок;
- возможность взаимодействия со стандартами обмена сообщениями XML и SOAP.

Первая цель регламентировалась требованиями удаленной обработки (Remoting), а вторая – веб-службами (Web Services). Работа по созданию одного механизма сериализации для достижения обеих целей была слишком сложной, поэтому в Microsoft решили построить два механизма: двоичный сериализатор и сериализатор XML.

Когда позже появилась инфраструктура Windows Communication Foundation (WCF), входящая в состав .NET Framework 3.0, частью цели была унификация Remoting и Web

Services. В итоге потребовался новый механизм сериализации, которым стал *сериализатор на основе контрактов данных*. Сериализатор на основе контрактов данных объединяет функциональные возможности предшествующих двух механизмов, касающиеся *обмена сообщениями (с возможностью взаимодействия)*. Однако за рамками такого контекста два более старых механизма по-прежнему важны.

### Сериализатор на основе контрактов данных

Сериализатор на основе контрактов данных является самым новым и наиболее универсальным из трех механизмов сериализации, к тому же он используется инфраструктурой WCF. Этот сериализатор особенно полезен в двух сценариях:

- при обмене информацией через протоколы обмена сообщениями, соответствующие отраслевым стандартам;
- когда необходимо обеспечить хорошую переносимость версий, а также возможность предохранения объектных ссылок.

Сериализатор на основе контрактов данных поддерживает модель *контрактов данных*, которая помогает отвязать низкоуровневые детали типов, подлежащих сериализации, от структуры сериализированных данных. В результате обеспечивается великолепная переносимость версий, что означает возможность десериализации данных, которые были сериализированы из более ранней или более поздней версии типа. Можно даже десериализировать типы, которые были переименованы или перемещены в другую сборку.

Сериализатор на основе контрактов данных способен справиться с большинством графов объектов, хотя он требует большего внимания, чем двоичный сериализатор. При наличии свободы в выборе структуры XML он также может применяться в качестве универсального инструмента для чтения/записи XML-файлов. (Если необходимо хранить данные в атрибутах или иметь дело с XML-элементами, расположенными в случайном порядке, то сериализатор на основе контрактов данных использовать нельзя.)

### Двоичный сериализатор

Механизм двоичной сериализации прост в применении, хорошо автоматизирован и повсеместно поддерживается внутри .NET Framework. Двоичная сериализация используется инфраструктурой Remoting, в том числе и при взаимодействии между двумя доменами приложений внутри одного процесса (как показано в главе 24).

Двоичный сериализатор хорошо автоматизирован: довольно часто единственный атрибут – все, что требуется для того, чтобы сделать сложный тип полностью сериализируемым. Двоичный сериализатор также работает быстрее сериализатора на основе контрактов данных, когда требуется высокая точность типов. Тем не менее, он тесно связывает внутреннюю структуру типа с форматом сериализированных данных, приводя в результате к плохой переносимости версий. (До выхода .NET Framework 2.0 даже добавление простого поля было изменением, нарушающим совместимость между версиями.) Механизм двоичной сериализации также не предназначен для выпуска XML-кода, хотя он предлагает форматер для обмена сообщениями на основе SOAP, который обеспечивает ограниченную возможность взаимодействия с простыми типами.

### `XmlSerializer`

Механизм сериализации на основе XML может генерировать *только* XML, и по сравнению с другими механизмами менее функционален при сохранении и восстановлении сложного графа объектов (он не позволяет восстанавливать разделяемые

объектные ссылки). Однако среди трех механизмов он обладает наибольшей гибкостью в следовании произвольной структуре XML. Например, можно выбирать, во что должны сериализироваться свойства – в элементы или в атрибуты, и обрабатывать внешний XML-элемент коллекции. Механизм сериализации XML также поддерживает великолепную переносимость версий.

Класс `XmlSerializer` применяется веб-службами ASMX.

### `IXmlSerializable`

Реализация интерфейса `IXmlSerializable` означает самостоятельное выполнение сериализации с помощью классов `XmlReader` и `XmlWriter`. Интерфейс `IXmlSerializable` распознается как классом `XmlSerializer`, так и сериализатором на основе контрактов данных, поэтому его можно избирательно использовать для более сложных типов. (Он также может применяться напрямую инфраструктурой WCF и веб-службами ASMX.) Классы `XmlReader` и `XmlWriter` были подробно описаны в главе 11.

## Форматеры

Вывод сериализатора на основе контрактов данных и двоичного сериализатора оформляется с помощью подключаемого *форматера.* Роль форматера одинакова в обоих механизмах сериализации, хотя для выполнения работы они используют совершенно разные классы.

Форматер приводит форму финального представления в соответствие с конкретной средой или контекстом сериализации. В общем случае можно выбирать между форматером XML и двоичным форматером. Форматер XML предназначен для работы внутри контекста средства чтения/записи XML, текстового файла/потока или пакета обмена сообщениями SOAP. Двоичный форматер предназначен для работы в контексте, где будут применяться произвольные потоки байтов – как правило, файл/поток или патентованный пакет обмена сообщениями. Двоичный вывод по размерам обычно меньше, чем XML – иногда значительно.

Понятие "двоичный" в контексте форматера не имеет отношения к механизму "двоичной" сериализации. Каждый из двух механизмов поставляется с форматером XML и двоичным форматером!

Теоретически механизмы не связаны со своими форматерами. На практике проектное решение каждого механизма согласовано с одним видом форматера. Сериализатор на основе контрактов данных согласован с требованиями взаимодействия при обмене сообщениями XML. Это хорошо для форматера XML, но означает, что его двоичный форматер не всегда настолько полезен, как можно было бы ожидать. Напротив, механизм двоичной сериализации предоставляет относительно неплохой двоичный форматер, но его форматер XML существенно ограничен, предлагая только грубую возможность взаимодействия с SOAP.

## Сравнение явной и неявной сериализации

Сериализация и десериализация могут быть инициированы двумя способами.

Первый способ предусматривает *явное* запрашивание сериализации и десериализации конкретного объекта. При явной сериализации или десериализации выбирается механизм и форматер.

В противоположность явной сериализации *неявная* сериализация запускается инфраструктурой .NET Framework, что происходит в следующих случаях:

- сериализатор рекурсивно сериализирует дочерний объект;
- используется средство, которое полагается на сериализацию, такое как WCF, Remoting или Web Services.

Инфраструктура WCF всегда применяет сериализатор на основе контрактов данных, хотя она может взаимодействовать с атрибутами и интерфейсами других механизмов.

Инфраструктура Remoting всегда имеет дело с механизмом двоичной сериализации.

Инфраструктура Web Services всегда работает с классом `XmlSerializer`.

# Сериализатор на основе контрактов данных

Использование сериализатора на основе контрактов данных предусматривает выполнение следующих базовых шагов.

1. Решить, какой класс применять — `DataContractSerializer` или `NetDataContractSerializer`.
2. Декорировать сериализируемые типы и члены с помощью атрибутов `[DataContract]` и `[DataMember]` соответственно.
3. Создать экземпляр сериализатора и вызвать его метод `WriteObject` или `ReadObject`.

В случае выбора `DataContractSerializer` также понадобится зарегистрировать "известные типы" (подтипы, которые тоже будут сериализироваться) и принять решение по поводу предохранения объектных ссылок.

Может еще возникнуть необходимость в специальном действии для обеспечения надлежащей сериализации коллекций.

Типы для сериализатора на основе контрактов данных определены в пространстве имен `System.Runtime.Serialization` внутри сборки с таким же именем.

## Сравнение `DataContractSerializer` и `NetDataContractSerializer`

Доступны два сериализатора на основе контрактов данных.

- `DataContractSerializer`. Обеспечивает слабую привязку типов .NET к типам контрактов данных.
- `NetDataContractSerializer`. Осуществляет тесную привязку типов .NET к типам контрактов данных.

Класс `DataContractSerializer` может генерировать совместимый со стандартами XML-код, обладающий возможностью взаимодействия, например:

```
<Person xmlns="...">
  ...
</Person>
```

Однако он требует предварительной явной регистрации сериализируемых подтипов, чтобы иметь возможность сопоставления имени контракта данных, такого как Person, с корректным типом .NET. Классу NetDataContractSerializer подобная помощь не нужна, т.к. он записывает полные имена типов и сборок для сериализируемых типов – почти как механизм двоичной сериализации:

```
<Person z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  ...
</Person>
```

Тем не менее, такой вывод является патентованным. При выполнении десериализации он также полагается на наличие определенного типа .NET в конкретном пространстве имен и сборке.

Если вы сохраняете граф объектов в "черный ящик", то можете выбрать любой из сериализаторов в зависимости от того, какие преимущества вам представляются важными. Если коммуникации производятся через WCF или выполняется чтение/запись XML-файла, тогда наиболее вероятно, что вам понадобится класс DataContractSerializer.

Еще одна разница между указанными двумя сериализаторами связана с тем, что NetDataContractSerializer всегда предохраняет эквивалентность ссылок, а DataContractSerializer делает это только по требованию.

Все упомянутые темы будут подробно рассматриваться в последующих разделах.

## Использование сериализаторов

Следующий шаг после выбора сериализатора – присоединение атрибутов к типам и членам, которые необходимо сериализировать. Необходимо предпринять минимум следующие действия:

- добавить атрибут [DataContract] к каждому типу;
- добавить атрибут [DataMember] к каждому члену, который должен быть включен.

Ниже приведен пример:

```
namespace SerialTest
{
  [DataContract] public class Person
  {
    [DataMember] public string Name;
    [DataMember] public int Age;
  }
}
```

Таких атрибутов вполне достаточно для того, чтобы сделать тип *неявно* сериализируемым посредством механизма сериализации на основе контрактов данных.

Затем объект можно явно сериализировать и десериализировать, создавая экземпляр класса DataContractSerializer или NetDataContractSerializer и вызывая метод WriteObject либо ReadObject:

```
Person p = new Person { Name = "Stacey", Age = 30 };

var ds = new DataContractSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  ds.WriteObject (s, p);  // Сериализировать
Person p2;
```

```
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) ds.ReadObject (s);  // Десериализировать

Console.WriteLine (p2.Name + " " + p2.Age);  // Stacey 30
```

Конструктор класса `DataContractSerializer` требует указания типа *корневого объекта* (типа объекта, который явно сериализируется). В отличие от него конструктор класса `NetDataContractSerializer` этого не требует:

```
var ns = new NetDataContractSerializer();
// В других отношениях класс NetDataContractSerializer
// используется точно так же, как DataContractSerializer.
...
```

Оба типа сериализаторов по умолчанию применяют форматер XML. При работе с классом `XmlWriter` можно запросить добавление в вывод отступов, улучшая читабельность:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

XmlWriterSettings settings = new XmlWriterSettings() { Indent = true };
using (XmlWriter w = XmlWriter.Create ("person.xml", settings))
  ds.WriteObject (w, p);

System.Diagnostics.Process.Start ("person.xml");
```

Ниже показан результат:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
      xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Age>30</Age>
  <Name>Stacey</Name>
</Person>
```

Имя XML-элемента `<Person>` отражает *имя контракта данных*, которое по умолчанию представляет собой имя типа .NET. Такое поведение можно переопределить и явно указать имя контракта данных следующим образом:

```
[DataContract (Name="Candidate")]
public class Person { ... }
```

Пространство имен XML отражает *пространство имен контракта данных*, которым по умолчанию является `http://schemas.datacontract.org/2004/07/`, а также пространство имен типа .NET. Поведение можно переопределить в аналогичной манере:

```
[DataContract (Namespace="http://oreilly.com/nutshell")]
public class Person { ... }
```

Указание имени и пространства имен разрывает связь между идентичностью контракта и именем типа .NET. Это позволяет гарантировать, что в случае проведения рефакторинга и изменения имени либо пространства имен типа сериализация не будет затронута.

Можно также переопределять имена данных-членов:

```
[DataContract (Name="Candidate", Namespace="http://oreilly.com/nutshell")]
public class Person
{
  [DataMember (Name="FirstName")]  public string Name;
  [DataMember (Name="ClaimedAge")] public int Age;
}
```

Вот как будет выглядеть вывод:

```
<?xml version="1.0" encoding="utf-8"?>
<Candidate xmlns="http://oreilly.com/nutshell"
         xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <ClaimedAge>30</ClaimedAge>
  <FirstName>Stacey</FirstName>
</Candidate>
```

Атрибут `[DataMember]` поддерживает поля и свойства — открытые и закрытые. Тип данных поля или свойства может быть одним из перечисленных ниже:

- любой примитивный тип;
- `DateTime`, `TimeSpan`, `Guid`, `Uri` или значение `Enum`;
- версии указанных выше типов, допускающие значение `null`;
- `byte[]` (сериализируется в XML с использованием кодировки Base64);
- любой "известный" тип, декорированный с помощью `DataContract`;
- любой тип, реализующий `IEnumerable` (как показано в разделе "Сериализация коллекций" далее в главе);
- любой тип, который снабжен атрибутом `[Serializable]` или реализует интерфейс `ISerializable` (как показано в разделе "Расширение контрактов данных" далее в главе);
- любой тип, реализующий `IXmlSerializable`.

### Указание двоичного форматера

Двоичный форматер можно применять с объектом `DataContractSerializer` или `NetDataContractSerializer`, например:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

var s = new MemoryStream();
using (XmlDictionaryWriter w = XmlDictionaryWriter.CreateBinaryWriter (s))
  ds.WriteObject (w, p);

var s2 = new MemoryStream (s.ToArray());
Person p2;
using (XmlDictionaryReader r = XmlDictionaryReader.CreateBinaryReader (s2,
                                 XmlDictionaryReaderQuotas.Max))
  p2 = (Person) ds.ReadObject (r);
```

Объем вывода варьируется между немного меньшим, чем при получении из форматера XML, и существенно меньшим, если типы содержат крупные массивы.

## Сериализация подклассов

Для поддержки сериализации подклассов с помощью `NetDataContractSerializer` никаких специальных усилий прикладывать не придется. Единственное требование заключается в том, что подклассы должны иметь атрибут `DataContract`. Сериализатор будет записывать полностью определенные имена действительных типов, которые он сериализирует, следующим образом:

```
<Person ... z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
```

Тем не менее, класс DataContractSerializer должен быть информирован обо всех подтипах, которые ему предстоит сериализировать или десериализировать. В целях иллюстрации предположим, что мы создаем подклассы Person, как показано ниже:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
}
[DataContract] public class Student : Person { }
[DataContract] public class Teacher : Person { }
```

и затем реализуем метод для клонирования экземпляра Person:

```
static Person DeepClone (Person p)
{
  var ds = new DataContractSerializer (typeof (Person));
  MemoryStream stream = new MemoryStream();
  ds.WriteObject (stream, p);
  stream.Position = 0;
  return (Person) ds.ReadObject (stream);
}
```

который вызываем так:

```
Person  person  = new Person  { Name = "Stacey", Age = 30 };
Student student = new Student { Name = "Stacey", Age = 30 };
Teacher teacher = new Teacher { Name = "Stacey", Age = 30 };

Person  p2 =           DeepClone (person);   // Нормально
Student s2 = (Student) DeepClone (student);  // Генерируется исключение
                                             // SerializationException
Teacher t2 = (Teacher) DeepClone (teacher);  // Генерируется исключение
                                             // SerializationException
```

Метод DeepClone работает, когда он вызывается с объектом Person, но приводит к генерации исключения при вызове с объектом Student или Teacher, поскольку десериализатор не имеет возможности узнать, в какой тип .NET (или сборку) должно быть преобразовано имя "Student" или "Teacher". Это также способствует обеспечению безопасности в том, что предотвращает десериализацию непредвиденных типов.

Решение предусматривает указание всех разрешенных, или "известных", подтипов. Такое действие можно предпринять либо при конструировании экземпляра DataContractSerializer:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (Student), typeof (Teacher) } );
```

либо в самом типе с помощью атрибута KnownType:

```
[DataContract, KnownType (typeof (Student)), KnownType (typeof (Teacher))]
public class Person
...
```

Ниже показано, как теперь будет выглядеть сериализированный объект Student:

```
<Person xmlns="..."
      xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
      i:type="Student" >
  ...
<Person>
```

Из-за того, что в качестве корневого типа указан Person, корневой элемент по-прежнему имеет такое имя. Действительный подкласс описан отдельно – в атрибуте type.

Класс NetDataContractSerializer оказывает высокое воздействие на производительность при сериализации подтипов, причем с любым форматером. Ситуация выглядит так, будто бы, столкнувшись с подтипом, форматер должен остановиться и подумать некоторое время!

Производительность сериализации имеет значение на сервере приложений, который обрабатывает множество параллельных запросов.

## Объектные ссылки

Ссылки на другие объекты также сериализируются. Рассмотрим следующие классы:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
  [DataMember] public Address HomeAddress;
}
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Ниже показан результат их сериализации в XML с использованием класса DataContractSerializer:

```
<Person...>
  <Age>...</Age>
  <HomeAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </HomeAddress>
  <Name>...</Name>
</Person>
```

Написанный в предыдущем разделе метод DeepClone клонировал бы также и член HomeAddress – что отличает его от простого метода MemberwiseClone.

Если работа производится с классом DataContractSerializer, то при создании подклассов Address применимы те же самые правила, как и при создании подклассов корневого типа. Таким образом, если мы определим, например, класс USAddress:

```
[DataContract]
public class USAddress : Address { }
```

и присвоим его экземпляр объекту Person:

```
Person p = new Person { Name = "John", Age = 30 };
p.HomeAddress = new USAddress { Street="Fawcett St", Postcode="02138" };
```

тогда объект p не сможет быть сериализирован. Решение предусматривает либо применение атрибута KnownType к Address:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

либо сообщение экземпляру DataContractSerializer о классе USAddress во время конструирования:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (USAddress) } );
```

(Сообщать о классе Address нет необходимости, потому что он является объявленным типом члена HomeAddress.)

### Предохранение объектных ссылок

Класс NetDataContractSerializer всегда предохраняет эквивалентность ссылок. Класс DataContractSerializer этого не делает без специального запроса.

Другими словами, если на один и тот же объект имеются ссылки в двух разных местах, то DataContractSerializer обычно записывает его дважды. Таким образом, если модифицировать предыдущий пример, чтобы класс Person также хранил рабочий адрес:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address HomeAddress, WorkAddress;
}
```

и затем сериализировать его экземпляр, как показано ниже:

```
Person p = new Person { Name = "Stacey", Age = 30 };
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
p.WorkAddress = p.HomeAddress;
```

тогда в XML можно будет увидеть, что те же самые детали адреса встречаются два раза:

```
...
<HomeAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</HomeAddress>
...
<WorkAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</WorkAddress>
```

При последующей десериализации приведенного XML-кода WorkAddress и HomeAddress будут разными объектами. Преимущество такой системы связано с тем, что она позволяет сохранить XML простым и совместимым со стандартами. Недостатки системы включают большой размер XML, утерю ссылочной целостности и невозможность справляться с циклическими ссылками.

Затребовать ссылочную целостность можно, указав true для аргумента preserveObjectReferences при конструировании DataContractSerializer:

```
var ds = new DataContractSerializer (typeof (Person),
                                     null, 1000, false, true, null);
```

Когда preserveObjectReferences равно true, третий аргумент является обязательным: он устанавливает максимальное количество объектных ссылок, которые сериализатор должен отслеживать. Если заданное количество будет превышено, тогда сериализатор сгенерирует исключение (и тем самым предотвратит возможность атаки типа отказа в обслуживании через поток, сконструированный в злонамеренных целях).

Вот как затем будет выглядеть XML для объекта Person с одинаковыми домашним и рабочим адресами:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
      xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
      z:Id="1">
  <Age>30</Age>
  <HomeAddress z:Id="2">
    <Postcode z:Id="3">6020</Postcode>
    <Street z:Id="4">Odo St</Street>
  </HomeAddress>
  <Name z:Id="5">Stacey</Name>
  <WorkAddress z:Ref="2" i:nil="true" />
</Person>
```

Платой будет сокращение возможностей взаимодействия (обратите внимание на патентованное пространство имен для атрибутов Id и Ref).

## Переносимость версий

Данные-члены можно добавлять и удалять, не нарушая прямой или обратной совместимости. По умолчанию десериализаторы на основе контрактов данных обладают следующими особенностями:

- пропускают данные, для которых в типе отсутствует атрибут [DataMember];
- не жалуются, если в потоке сериализации отсутствуют данные, для которых в типе предусмотрен атрибут [DataMember].

Вместо пропуска нераспознанных данных десериализатору можно сообщить о необходимости сохранить нераспознанные данные-члены в "черном ящике" и воспроизвести их позже, когда тип будет сериализироваться повторно. Это позволяет корректно обходиться с данными, которые были сериализированы более поздней версией типа. Для активизации такой возможности необходимо реализовать интерфейс IExtensibleDataObject. Указанный интерфейс на самом деле можно было назвать поставщиком "черного ящика" (скажем, IBlackBoxProvider). Он требует реализации единственного свойства, предназначенного для получения/установки "черного ящика":

```
[DataContract] public class Person : IExtensibleDataObject {
  [DataMember] public string Name;
  [DataMember] public int Age;
  ExtensionDataObject IExtensibleDataObject.ExtensionData { get; set; }
}
```

### Обязательные члены

Если член является жизненно важным для типа, то с помощью аргумента IsRequired можно потребовать его обязательного присутствия:

```
[DataMember (IsRequired=true)] public int ID;
```

Если такой член отсутствует, тогда при десериализации сгенерируется исключение.

## Упорядочение членов

Сериализаторы на основе контрактов данных крайне придирчивы к порядку следования членов. На самом деле десериализаторы *пропускают любые члены, которые считаются неупорядоченными.*

При сериализации члены записываются в описанном далее порядке.

1. Порядок от базового класса к подклассу.
2. Порядок от низких значений `Order` к высоким значениям `Order` (для данных-членов с установленным аргументом `Order`).
3. Алфавитный порядок (с использованием *ординального* сравнения строк).

Таким образом, в предшествующих примерах `Age` будет находиться перед `Name`. В следующем примере `Name` находится перед `Age`:

```
[DataContract] public class Person
{
  [DataMember (Order=0)] public string Name;
  [DataMember (Order=1)] public int Age;
}
```

При наличии у класса `Person` базового класса сначала сериализировались бы все данные-члены базового класса.

Основная причина для указания порядка — соответствие определенной схеме XML. Порядок следования XML-элементов совпадает с порядком следования данных-членов.

Если возможность взаимодействия с чем-то другим не нужна, то простейший подход заключается в том, чтобы *не* указывать значения `Order` для членов и полагаться чисто на упорядочение по алфавиту. Тогда в случае добавления и удаления членов расхождение между сериализацией и десериализацией никогда не возникнет. Единственная ситуация, когда произойдет нестыковка — перемещение члена между базовым классом и подклассом.

## Пустые значения и `null`

Существуют два способа обработки члена с пустым значением или `null`.

1. Явно записать пустое значение или `null` (стандартный способ).
2. Не помещать член в вывод сериализации.

В XML явное значение `null` выглядит так:

```
<Person xmlns="..."
      xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Name i:nil="true" />
</Person>
```

Записывание членов с пустыми значениями или `null` может приводить к бесполезному расходу пространства, особенно в случае типов с множеством полей или свойств, которые обычно остаются пустыми. Более важно то, что может возникнуть необходимость следовать XML-схеме, которая ожидает применения необязательных элементов (скажем, `minOccurs="0"`), а не значений `nil`.

Проинструктировать сериализатор о том, что он не должен выдавать данные-члены для пустых значений или null, можно следующим образом:

```
[DataContract] public class Person
{
  [DataMember (EmitDefaultValue=false)] public string Name;
  [DataMember (EmitDefaultValue=false)] public int Age;
}
```

Член Name будет пропущен, если его значение равно null, а член Age — если его значением является 0 (стандартное значение для типа int). В случае объявления Age с типом int, допускающим null, данный член будет пропускаться тогда (и только тогда), когда его значением окажется null.

Десериализатор на основе контрактов данных при восстановлении объекта пропускает конструкторы и инициализаторы полей типа. Это позволяет пропускать данные-члены, как было описано выше, не разрушая поля, которым присвоены нестандартные значения, из-за выполнения инициализатора или конструктора. В целях иллюстрации предположим, что в качестве стандартного значения для Age в Person выбрано 30:

```
[DataMember (EmitDefaultValue=false)]
public int Age = 30;
```

А теперь представим, что мы создаем объект Person, явно переустанавливаем его поле Age из 30 в 0 и затем сериализируем его. Вывод не будет включать Age, поскольку 0 — стандартное значение для типа int. В итоге при десериализации поле Age будет проигнорировано и останется со своим стандартным значением, которое по счастливой случайности равно 0, учитывая, что инициализаторы полей и конструкторы были пропущены.

# Контракты данных и коллекции

Сериализаторы на основе контрактов данных могут сохранять и повторно наполнять любую перечислимую коллекцию. Например, пусть класс Person определен со списком List<> адресов:

```
[DataContract] public class Person
{
  ...
  [DataMember] public List<Address> Addresses;
}
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Ниже показан результат сериализации объекта Person с двумя адресами:

```
<Person ...>
  ...
  <Addresses>
    <Address>
      <Postcode>6020</Postcode>
      <Street>Odo St</Street>
    </Address>
```

```
    <Address>
      <Postcode>6152</Postcode>
      <Street>Comer St</Street>
    </Address>
  </Addresses>
  ...
</Person>
```

Обратите внимание, что сериализатор не кодирует информацию о конкретном *типе* коллекции, которую он сериализирует. Если бы поле `Addresses` имело тип `Address[]`, то вывод был бы идентичным. Это позволяет изменять тип коллекции между сериализацией и десериализацией, не приводя к ошибке.

Тем не менее, иногда требуется, чтобы коллекция была более специфичного типа, чем указанный. Самым крайним примером является вариант с интерфейсами:

```
[DataMember] public IList<Address> Addresses;
```

Данный член сериализируется корректно (как и ранее), но во время десериализации возникнет проблема. У десериализатора нет никакой возможности узнать, объект какого конкретного типа должен быть создан, так что он выбирает простейший вариант — массив. Десериализатор придерживается такой стратегии, даже когда вы инициализируете поле с другим конкретным типом:

```
[DataMember] public IList<Address> Addresses = new List<Address>();
```

(Вспомните, что десериализатор пропускает инициализаторы полей.) Обойти проблему можно, сделав член закрытым полем и добавив открытое свойство для доступа к нему:

```
[DataMember (Name="Addresses")] List<Address> _addresses;

public IList<Address> Addresses { get { return _addresses; } }
```

Вполне вероятно, в нетривиальных приложениях вы будете в любом случае использовать свойства в подобной манере. Единственный необычный момент здесь касается пометки в качестве члена данных закрытого поля, а не открытого свойства.

## Элементы коллекции, являющиеся подклассами

Сериализатор прозрачно обрабатывает элементы коллекции, являющиеся подклассами. Допустимые подтипы должны объявляться точно так же, как если бы они применялись в любом другом месте:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
public class USAddress : Address { }
```

Добавление `USAddress` в список адресов `Person` приводит к генерации XML следующего вида:

```
...
  <Addresses>
    <Address i:type="USAddress">
      <Postcode>02138</Postcode>
      <Street>Fawcett St</Street>
    </Address>
  </Addresses>
```

## Настройка имен коллекции и элементов

В случае создания подкласса от самого класса коллекции XML-имя, которое используется для описания каждого элемента, можно настраивать, присоединяя атрибут CollectionDataContract:

```
[CollectionDataContract (ItemName="Residence")]
public class AddressList : Collection<Address> { }

[DataContract] public class Person
{
  ...
  [DataMember] public AddressList Addresses;
}
```

Вот результат:

```
...
  <Addresses>
    <Residence>
      <Postcode>6020</Postcode
      <Street>Odo St</Street>
    </Residence>
    ...
```

Атрибут CollectionDataContract также позволяет указывать аргументы Namespace и Name. Последний аргумент не применяется, когда коллекция сериализирована как свойство другого объекта (вроде того, что было в приведенном примере), но используется в случае, если коллекция сериализируется как корневой объект.

Атрибут CollectionDataContract можно также применять для управления сериализацией словарей:

```
[CollectionDataContract (ItemName="Entry",
                         KeyName="Kind",
                       ValueName="Number")]
public class PhoneNumberList : Dictionary <string, string> { }

[DataContract] public class Person
{
  ...
  [DataMember] public PhoneNumberList PhoneNumbers;
}
```

Ниже представлен результат:

```
...
  <PhoneNumbers>
    <Entry>
      <Kind>Home</Kind>
      <Number>08 1234 5678</Number>
    </Entry>
    <Entry>
      <Kind>Mobile</Kind>
      <Number>040 8765 4321</Number>
    </Entry>
  </PhoneNumbers>
```

# Расширение контрактов данных

В настоящем разделе будет показано, как можно расширять функциональные возможности сериализатора на основе контрактов данных с помощью ловушек сериализации, атрибута `[Serializable]` и интерфейса `IXmlSerializable`.

## Ловушки сериализации и десериализации

Можно затребовать, чтобы до или после сериализации выполнялся специальный метод, пометив его одним из перечисленных ниже атрибутов.

| | |
|---|---|
| `[OnSerializing]` | Указывает метод для вызова *перед* сериализацией. |
| `[OnSerialized]` | Указывает метод для вызова *после* сериализации. Аналогичные атрибуты поддерживаются и для десериализации. |
| `[OnDeserializing]` | Указывает метод для вызова *перед* десериализацией. |
| `[OnDeserialized]` | Указывает метод для вызова *после* десериализации. |

Специальный метод должен иметь единственный параметр типа `StreamingContext`. Данный параметр обязателен для обеспечения согласованности с механизмом двоичной сериализации; сериализатор на основе контрактов данных его не использует.

Атрибуты `[OnSerializing]` и `[OnDeserialized]` пригодны для обработки членов, которые выходят за рамки возможностей механизма сериализации на основе контрактов данных, таких как коллекция, несущая дополнительную полезную нагрузку или не реализующая стандартные интерфейсы. Вот как выглядит базовый подход:

```
[DataContract] public class Person
{
  public SerializationUnfriendlyType Addresses;

  [DataMember (Name="Addresses")]
  SerializationFriendlyType _serializationFriendlyAddresses;

  [OnSerializing]
  void PrepareForSerialization (StreamingContext sc)
  {
    // Копировать Addresses в _serializationFriendlyAddresses
    // ...
  }

  [OnDeserialized]
  void CompleteDeserialization (StreamingContext sc)
  {
    // Копировать _serializationFriendlyAddresses в Addresses
    // ...
  }
}
```

Метод `[OnSerializing]` может также применяться для условной сериализации полей:

```
public DateTime DateOfBirth;

[DataMember] public bool Confidential;

[DataMember (Name="DateOfBirth", EmitDefaultValue=false)]
DateTime? _tempDateOfBirth;

[OnSerializing]
```

```
void PrepareForSerialization (StreamingContext sc)
{
  if (Confidential)
    _tempDateOfBirth = DateOfBirth;
  else
    _tempDateOfBirth = null;
}
```

Вспомните, что десериализаторы на основе контрактов данных пропускают инициализаторы полей и конструкторы. Метод `[OnDeserializing]` действует в качестве псевдоконструктора для десериализации, и он удобен для инициализации полей, исключенных из сериализации:

```
[DataContract] public class Test
{
  bool _editable = true;

  public Test() { _editable = true; }

  [OnDeserializing]
  void Init (StreamingContext sc)
  {
    _editable = true;
  }
}
```

Если бы не метод `Init`, то поле `_editable` в десериализированном экземпляре `Test` содержало бы значение `false`, несмотря на две попытки сделать его равным `true`.

Методы, декорированные упомянутыми четырьмя атрибутами, могут быть закрытыми. Если должны участвовать подтипы, то они могут определять собственные методы с теми же самыми атрибутами, и такие методы также будут выполнены.

## Возможность взаимодействия с помощью `[Serializable]`

Сериализатор на основе контрактов данных может также сериализировать типы, помеченные с помощью атрибутов и интерфейсов механизма двоичной сериализации. Такая возможность важна, поскольку поддержка механизма двоичной сериализации широко использовалась в коде, написанном до выхода версии .NET Framework 3.0, включая и саму инфраструктуру .NET Framework!

Пометить тип как сериализируемый для механизма двоичной сериализации можно посредством:

- атрибута `[Serializable]`;
- реализации интерфейса `ISerializable`.

Возможность двоичного взаимодействия полезна при сериализации существующих типов, а также новых типов, которые нуждаются в поддержке обоих механизмов. Она также предоставляет еще одно средство расширения возможностей сериализатора на основе контрактов данных, потому что интерфейс `ISerializable` механизма двоичной сериализации является более гибким, чем атрибуты контрактов данных. К сожалению, сериализатор на основе контрактов данных неэффективен в том, как он форматирует данные, добавленные через `ISerializable`.

В типе, для которого желательно извлечь лучшее из двух технологий, нельзя определять атрибуты для обоих механизмов. Иначе создалась бы проблема для таких типов, как `string` и `DateTime`, которые по историческим причинам не могут быть отделены от атрибутов механизма двоичной сериализации. Сериализатор на основе контрактов данных обходит указанную проблему за счет фильтрации базовых типов и их обработки специальным образом. Ко всем другим типам, помеченным для двоичной сериализации, сериализатор на основе контрактов данных применяет правила, похожие на те, которые использовал бы механизм двоичной сериализации. Это означает, что он учитывает атрибуты вроде `NonSerialized` или обращается к интерфейсу `ISerializable` в случае его реализации. Он не *переходит* на сам механизм двоичной сериализации, гарантируя тем самым, что вывод форматируется в таком же стиле, как если бы применялись атрибуты контрактов данных.

Типы, предназначенные для сериализации с помощью двоичного механизма, ожидают предохранения объектных ссылок. Активизировать его можно через класс `DataContractSerializer` (или за счет использования класса `NetDataContractSerializer`).

Правила для регистрации известных типов также применяются к объектам и подобъектам, которые сериализированы посредством механизма двоичной сериализации.

В следующем примере демонстрируется класс с членом `[Serializable]`:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address MailingAddress;
}
[Serializable] public class Address
{
  public string Postcode, Street;
}
```

Вот результат его сериализации:

```
<Person ...>
  ...
  <MailingAddress>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </MailingAddress>
  ...
```

Если бы класс `Address` реализовывал интерфейс `ISerializable`, то результат оказался бы не настолько эффективно сформатированным:

```
<MailingAddress>
  <Street xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">str</Street>
  <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">pcode</Postcode>
</MailingAddress>
```

### Возможность взаимодействия с помощью `IXmlSerializable`

Ограничение сериализатора на основе контрактов данных заключается в том, что он предоставляет лишь небольшой контроль над структурой XML. На самом деле это может быть выгодно для приложений WCF, т.к. упрощается согласование инфраструктуры со стандартными протоколами обмена сообщениями.

Если необходим более точный контроль над XML, тогда можно реализовать интерфейс `IXmlSerializable` и затем использовать классы `XmlReader` и `XmlWriter` для чтения и записи XML вручную. Сериализатор на основе контрактов данных позволяет поступать так только с типами, для которых подобный уровень контроля является обязательным. Мы опишем интерфейс `IXmlSerializable` в последнем разделе главы.

## Двоичный сериализатор

Механизм двоичной сериализации неявно применяется инфраструктурой Remoting. Он также может использоваться для решения таких задач, как сохранение и восстановление объектов с диска. Двоичная сериализация хорошо автоматизирована и может обрабатывать сложные графы объектов с минимальным вмешательством. Однако она не доступна в приложениях Windows Store.

Сделать тип поддерживающим двоичную сериализацию можно двумя путями. Первый из них основан на атрибутах, а второй предусматривает реализацию интерфейса `ISerializable`. Добавление атрибутов проще, но реализация `ISerializable` предлагает более высокую гибкость. Интерфейс `ISerializable` обычно реализуется для достижения следующих целей:

- динамическое управление тем, что сериализируется;
- обеспечение удобства создания подклассов сериализируемого типа другими потребителями.

### Начало работы

Тип делается сериализируемым с помощью единственного атрибута:

```
[Serializable] public sealed class Person
{
  public string Name;
  public int Age;
}
```

Атрибут `[Serializable]` инструктирует сериализатор о необходимости включения всех полей в данном типе, что касается как закрытых, так и открытых полей (но не свойств). Каждое поле само должно допускать сериализацию, иначе сгенерируется исключение. Примитивные типы .NET, такие как `string` и `int`, поддерживают сериализацию (подобно многим другим типам .NET).

Атрибут `Serializable` не наследуется, а потому подклассы не будут автоматически сериализируемыми, если их не пометить указанным атрибутом.

В случае автоматических свойств механизм двоичной сериализации сериализирует лежащие в основе поля, генерируемые компилятором. К сожалению, имена таких полей могут изменяться при перекомпиляции типа, нарушая совместимость с существующими сериализированными данными. Обойти проблему можно либо за счет устранения автоматических свойств в типах `[Serializable]`, либо путем реализации интерфейса `ISerializable`.

Чтобы сериализировать экземпляр `Person`, необходимо создать объект форматера и вызвать метод `Serialize`. Для применения с механизмом двоичной сериализации предназначены два форматера.

- `BinaryFormatter`. Из двух это более эффективный форматер, который генерирует вывод меньшего объема за короткое время. Он определен в пространстве имен `System.Runtime.Serialization.Formatters.Binary`.
- `SoapFormatter`. Данный форматер поддерживает базовый обмен сообщениями в стиле SOAP, когда используется вместе с Remoting. Он определен в пространстве имен `System.Runtime.Serialization.Formatters.Soap`.

Класс `BinaryFormatter` находится в `mscorlib`, а `SoapFormatter` — в `System.Runtime.Serialization.Formatters.Soap.dll`.

Класс `SoapFormatter` менее функционален, чем `BinaryFormatter`. Класс `SoapFormatter` не поддерживает обобщенные типы или фильтрацию несовместимых данных, которая необходима при сериализации, обеспечивающей переносимость версий.

Указанные два форматера применяются одинаково. Следующий код выполняет сериализацию объекта `Person` с помощью `BinaryFormatter`:

```
Person p = new Person() { Name = "George", Age = 25 };
IFormatter formatter = new BinaryFormatter();
using (FileStream s = File.Create ("serialized.bin"))
  formatter.Serialize (s, p);
```

Все данные, необходимые для реконструирования объекта `Person`, записываются в файл `serialized.bin`. Метод `Deserialize` восстанавливает объект:

```
using (FileStream s = File.OpenRead ("serialized.bin"))
{
  Person p2 = (Person) formatter.Deserialize (s);
  Console.WriteLine (p2.Name + " " + p.Age);     // George 25
}
```

При воссоздании объектов десериализатор пропускает все конструкторы. "За кулисами" для выполнения такой работы он вызывает метод `FormatterServices.GetUninitializedObject`. Его можно вызывать напрямую для реализации черновых шаблонов проектирования.

Сериализированные данные включают полные сведения о типе и сборке, поэтому если попытаться привести результат десериализации к совпадающему типу `Person` из другой сборки, то возникнет ошибка. Десериализатор восстанавливает объектные ссылки полностью в их исходном состоянии. Это касается и коллекций, которые трак-

туются просто как сериализируемые объекты, похожие на любые другие (все коллекции, определенные в пространствах имен `System.Collections.*`, помечены как сериализируемые).

Механизм двоичной сериализации может обрабатывать крупные и сложные графы объектов, не требуя специальной поддержки (кроме обеспечения возможности сериализации всех участвующих членов). Единственный момент, к которому следует относиться осторожно – тот факт, что производительность сериализатора снижается пропорционально количеству ссылок в графе объектов. В итоге может возникнуть проблема на сервере Remoting, который должен обрабатывать много параллельных запросов.

# Атрибуты двоичной сериализации

## [NonSerialized]

В отличие от контрактов данных, поддерживающих политику *включения* (opt-in) при сериализации полей, механизм двоичной сериализации реализует политику *отключения* (opt-out). Поля, которые не должны сериализироваться, такие как используемые для временных вычислений или для хранения файловых либо оконных дескрипторов, потребуется явно пометить с помощью атрибута `[NonSerialized]`:

```
[Serializable] public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;

  // Поле Age может быть вычислено, поэтому в сериализации не нуждается.
  [NonSerialized] public int Age;
}
```

Атрибут инструктирует сериализатор игнорировать член `Age`.

Несериализированные члены при десериализации всегда получают пустое значение или `null`, даже если инициализаторы полей или конструкторы устанавливают их по-другому.

## [OnDeserializing] и [OnDeserialized]

Десериализация пропускает все обычные конструкторы, а также инициализаторы полей. Это не особенно важно, когда в сериализации принимают участие все поля, но может привести к проблемам, если некоторые поля исключены через `[NonSerialized]`. В целях иллюстрации добавим в класс `Person` поле типа `bool` по имени `Valid`:

```
public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;

  [NonSerialized] public int Age;
  [NonSerialized] public bool Valid = true;

  public Person() { Valid = true; }
}
```

В десериализированном объекте Person поле Valid будет иметь значение false — несмотря на его установку в true внутри конструктора и инициализатора поля.

Решение здесь такое же, как и в случае сериализатора на основе контрактов данных: нужно определить специальный "конструктор" десериализации с помощью атрибута [OnDeserializing]. Метод, помеченный таким атрибутом, будет вызываться прямо перед десериализацией:

```
[OnDeserializing]
void OnDeserializing (StreamingContext context)
{
  Valid = true;
}
```

Можно было бы также написать метод [OnDeserialized] для обновления вычисляемого поля Age (он запускается сразу *после* десериализации):

```
[OnDeserialized]
void OnDeserialized (StreamingContext context)
{
  TimeSpan ts = DateTime.Now - DateOfBirth;
  Age = ts.Days / 365;  // Примерный возраст в годах
}
```

## [OnSerializing] и [OnSerialized]

Механизм двоичной сериализации также поддерживает атрибуты [OnSerializing] и [OnSerialized]. С их помощью помечается метод, подлежащий выполнению соответственно до и после сериализации. Чтобы взглянуть, чем может быть полезен такой прием, мы определим класс Team, который содержит обобщенный список игроков:

```
[Serializable] public sealed class Team
{
  public string Name;
  public List<Person> Players = new List<Person>();
}
```

Класс Team корректно сериализируется и десериализируется с применением двоичного форматера, но не форматера SOAP. Причина связана с неочевидным ограничением: форматер SOAP отказывается сериализировать обобщенные типы! Простейшее решение предусматривает преобразование списка Players в массив непосредственно перед сериализацией и обратное преобразование массива в обобщенный List после десериализации. Чтобы это заработало, мы можем добавить еще одно поле для хранения массива, пометить исходное поле Players как [NonSerialized] и затем написать код преобразования следующим образом:

```
[Serializable] public sealed class Team
{
  public string Name;
  Person[] _playersToSerialize;

  [NonSerialized] public List<Person> Players = new List<Person>();

  [OnSerializing]
  void OnSerializing (StreamingContext context)
  {
    _playersToSerialize = Players.ToArray();
  }
```

```
  [OnSerialized]
  void OnSerialized (StreamingContext context)
  {
    _playersToSerialize = null; //Разрешить массиву быть освобожденным из памяти
  }
  [OnDeserialized]
  void OnDeserialized (StreamingContext context)
  {
    Players = new List<Person> (_playersToSerialize);
  }
}
```

## [OptionalField] и поддержка версий

По умолчанию добавление поля нарушает совместимость с данными, которые уже сериализированы, если только не пометить новое поле атрибутом `[OptionalField]`.

В целях иллюстрации предположим, что мы начали с класса `Person`, который имеет только одно поле. Назовем его версией 1:

```
[Serializable] public sealed class Person  // Версия 1
{
  public string Name;
}
```

Позже мы обнаруживаем, что необходимо второе поле, поэтому создаем версию 2 класса `Person`:

```
[Serializable] public sealed class Person  // Версия 2
{
  public string Name;
  public DateTime DateOfBirth;
}
```

Если два компьютера обменивались объектами `Person` через инфраструктуру Remoting, то десериализация не будет нормально работать до тех пор, пока оба компьютера не обновятся до версии 2 *точно в одно и то же время*. Обойти проблему позволяет атрибут `OptionalField`:

```
[Serializable] public sealed class Person  // Версия 2 надежна
{
  public string Name;
  [OptionalField (VersionAdded = 2)] public DateTime DateOfBirth;
}
```

Атрибут `OptionalField` сообщает десериализатору о том, что он не должен паниковать, если в потоке данных не встречаются значения для `DateOfBirth`, а считать недостающее поле несериализированным. В конечном итоге поле `DateTime` оказывается пустым (в методе `[OnDeserializing]` ему можно присвоить другое значение).

Аргумент `VersionAdded` представляет собой целочисленное значение, которое инкрементируется при каждом добавлении полей к типу. Оно служит в качестве документации и никак не влияет на семантику сериализации.

Если надежность поддержки версий важна, тогда избегайте переименования и удаления, а также ретроспективного добавления атрибута `NonSerialized`. Никогда не изменяйте типы полей.

До сих пор мы были сосредоточены на проблеме обратной совместимости: когда десериализатор не может найти ожидаемое поле в потоке сериализации. Но при двухсторонних коммуникациях может также возникать проблема прямой совместимости, когда десериализатор обнаруживает постороннее поле и не знает, как его обработать. Двоичный форматер запрограммирован на автоматическое отбрасывание посторонних данных; форматер SOAP взамен генерирует исключение! Таким образом, если требуется обеспечение надежной поддержки версий при двухсторонних коммуникациях, должен использоваться двоичный форматер; иначе сериализацией придется управлять вручную, реализуя интерфейс `ISerializable`.

# Двоичная сериализация с помощью `ISerializable`

Реализация интерфейса `ISerializable` предоставляет типу полный контроль над прохождением его двоичной сериализации и десериализации.

Ниже показано определение интерфейса `ISerializable`:

```
public interface ISerializable
{
  void GetObjectData (SerializationInfo info, StreamingContext context);
}
```

Метод `GetObjectData` запускается при сериализации; его работа заключается в наполнении объекта `SerializationInfo` (словарь пар "имя/значение") данными из всех полей, которые необходимо сериализировать. Вот как можно реализовать метод `GetObjectData`, сериализирующий два поля с именами `Name` и `DateOfBirth`:

```
public virtual void GetObjectData (SerializationInfo info,
                                   StreamingContext context)
 {
   info.AddValue ("Name", Name);
   info.AddValue ("DateOfBirth", DateOfBirth);
 }
```

В приведенном примере мы решили именовать каждый элемент согласно соответствующему ему полю. Поступать так вовсе не обязательно; может применяться любое имя при условии, что при десериализации используется точно такое же имя. Сами значения могут иметь любой сериализируемый тип; при необходимости инфраструктура .NET Framework будет выполнять рекурсивную сериализацию. В словаре разрешено хранить значения `null`.

Неплохо объявить метод `GetObjectData` как `virtual` — если только класс не является `sealed`. Это позволит подклассам расширять сериализацию без необходимости в повторной реализации интерфейса `ISerializable`.

Класс `SerializationInfo` также содержит свойства, которые можно применять для управления типом и сборкой, куда экземпляр должен быть десериализирован. Параметр `StreamingContext` представляет собой структуру, содержащую помимо прочего значение перечисления, которое указывает, откуда поступает сериализированный экземпляр (диск, Remoting и т.д., хотя данное значение не всегда установлено).

В дополнение к реализации интерфейса `ISerializable` тип, управляющий собственной сериализацией, должен предоставить конструктор десериализации, который принимает такие же два параметра, как и конструктор `GetObjectData`. Конструктор может быть объявлен с любым уровнем доступа – исполняющая среда все равно его найдет. Обычно он объявляется как `protected`, так что подклассы могут его вызывать.

В следующем примере мы реализуем интерфейс `ISerializable` в классе `Team`. Когда дело доходит до обработки списка `List` игроков, мы сериализируем данные в виде массива, а не обобщенного списка, обеспечивая совместимость с форматером SOAP:

```
[Serializable] public class Team : ISerializable
{
  public string Name;
  public List<Person> Players;

  public virtual void GetObjectData (SerializationInfo si,
                                     StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    si.AddValue ("PlayerData", Players.ToArray());
  }

  public Team() {}

  protected Team (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");

    // Десериализировать Players в массив для соответствия сериализации:
    Person[] a = (Person[]) si.GetValue ("PlayerData", typeof (Person[]));

    // Сконструировать новый список List, используя этот массив:
    Players = new List<Person> (a);
  }
}
```

Для широко используемых типов в классе `SerializationInfo` есть типизированные методы `Get*`, такие как `GetString`, предназначенные для упрощения реализации конструкторов десериализации. В случае указания имени, для которого данные не существуют, генерируется исключение. Чаще всего это происходит, когда имеется несовпадение версий в коде, выполняющем сериализацию и десериализацию. Например, вы добавили дополнительное поле и не подумали о последствиях, которые вызовет десериализация старого экземпляра. Чтобы обойти такую проблему, можно поступить следующим образом:

- добавить обработку исключений к коду, который извлекает член, добавленный в последней версии;
- реализовать собственную систему нумерации версий, например:

```
public string MyNewField;

public virtual void GetObjectData (SerializationInfo si,
                                   StreamingContext sc)
{
  si.AddValue ("_version", 2);
  si.AddValue ("MyNewField", MyNewField);
  ...
}
```

```
protected Team (SerializationInfo si, StreamingContext sc)
{
  int version = si.GetInt32 ("_version");
  if (version >= 2) MyNewField = si.GetString ("MyNewField");
  ...
}
```

## Создание подклассов из сериализируемых классов

В предшествующих примерах классы, полагающиеся на атрибуты для сериализации, запечатывались с помощью sealed. Чтобы понять причину, рассмотрим следующую иерархию классов:

```
[Serializable] public class Person
{
  public string Name;
  public int Age;
}

[Serializable] public sealed class Student : Person
{
  public string Course;
}
```

В этом примере классы Person и Student являются сериализируемыми, и оба они задействуют стандартное поведение сериализации исполняющей среды, т.к. ни один из них не реализует интерфейс ISerializable.

Теперь предположим, что разработчик класса Person решил по какой-то причине реализовать интерфейс ISerializable и предоставить конструктор десериализации, чтобы управлять сериализацией Person. Новая версия Person может иметь такой вид:

```
[Serializable] public class Person : ISerializable
{
  public string Name;
  public int Age;

  public virtual void GetObjectData (SerializationInfo si,
                                     StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    si.AddValue ("Age", Age);
  }

  protected Person (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");
    Age = si.GetInt32 ("Age");
  }

  public Person() {}
}
```

Несмотря на возможность работы с экземплярами Person, внесенное изменение нарушает сериализацию экземпляров Student. Сериализация экземпляра Student выглядит успешной, однако поле Course в типе Student не сохраняется в потоке, поскольку реализации метода ISerializable.GetObjectData в классе Person ничего не известно о членах типа, производного от Student. Вдобавок десериализация

экземпляров Student генерирует исключение, потому что исполняющая среда ищет (безуспешно) конструктор десериализации в Student.

Решение продемонстрированной проблемы предусматривает реализацию интерфейса ISerializable с самого начала для сериализируемых классов, которые являются открытыми и незапечатанными. (Для классов internal это не настолько важно, т.к. подклассы можно легко модифицировать позже, если потребуется.)

Если мы начинаем с написания класса Person как в предыдущем примере, тогда класс Student должен быть реализован следующим образом:

```
[Serializable]
public class Student : Person
{
  public string Course;

  public override void GetObjectData (SerializationInfo si,
                                      StreamingContext sc)
  {
    base.GetObjectData (si, sc);
    si.AddValue ("Course", Course);
  }

  protected Student (SerializationInfo si, StreamingContext sc)
    : base (si, sc)
  {
    Course = si.GetString ("Course");
  }
  public Student() {}
}
```

# Сериализация XML

Инфраструктура .NET Framework предлагает в пространстве имен System.Xml.Serialization отдельный механизм сериализации XML под названием XmlSerializer. Он подходит для сериализации типов .NET в XML-файлы и также неявно применяется веб-службами ASMX.

Как и в случае механизма двоичной сериализации, на выбор доступны два подхода:

- добавить к типам атрибуты (определенные в System.Xml.Serialization);
- реализовать интерфейс IXmlSerializable.

Однако в отличие от механизма двоичной сериализации реализация интерфейса (т.е. IXmlSerializable) полностью избегает использования механизма, оставляя на разработчика самостоятельное написание кода сериализации с участием классов XmlReader и XmlWriter.

## Начало работы с сериализацией на основе атрибутов

Для применения класса XmlSerializer необходимо создать его экземпляр и вызвать метод Serialize или Deserialize с потоком (Stream) и интересующим объектом. В целях иллюстрации предположим, что определен следующий класс:

```
public class Person
{
  public string Name;
  public int Age;
}
```

Приведенный ниже код сохраняет объект Person в XML-файл и затем его восстанавливает:

```
Person p = new Person();
p.Name = "Stacey"; p.Age = 30;

XmlSerializer xs = new XmlSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  xs.Serialize (s, p);

Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) xs.Deserialize (s);

Console.WriteLine (p2.Name + " " + p2.Age);   // Stacey 30
```

Методы Serialize и Deserialize могут работать с объектами Stream, XmlWriter/XmlReader или TextWriter/TextReader. Ниже показан результирующий XML-код:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Stacey</Name>
  <Age>30</Age>
</Person>
```

Класс XmlSerializer способен сериализировать типы, не имеющие ни одного атрибута — такие как наш тип Person. По умолчанию он сериализирует *все открытые поля и свойства* типа. Исключить члены из числа сериализируемых можно посредством атрибута XmlIgnore:

```
public class Person
{
  ...
  [XmlIgnore] public DateTime DateOfBirth;
}
```

В отличие от двух других механизмов класс XmlSerializer не распознает атрибут [OnDeserializing], а вместо него при десериализации полагается на конструктор без параметров, генерируя исключение, если он отсутствует. (В показанном выше примере класс Person имеет *неявный* конструктор без параметров.) Это также означает выполнение инициализаторов полей перед десериализацией:

```
public class Person
{
  public bool Valid = true;  // Выполняется перед десериализацией
}
```

Хотя XmlSerializer может сериализировать почти любой тип, он распознает перечисленные далее типы и трактует их особым образом:

- примитивные типы, типы DateTime, TimeSpan, Guid, а также их версии, допускающие null;
- тип byte[] (который преобразуется с использованием кодировки Base64);
- тип XmlAttribute или XmlElement (его содержимое внедряется в поток);
- любой тип, реализующий интерфейс IXmlSerializable;
- любой тип коллекции.

Десериализатор является переносимым в плане версий: он не жалуется, если отсутствуют элементы или атрибуты, либо встречаются лишние данные.

### Атрибуты, имена и пространства имен

По умолчанию поля и свойства сериализируются в XML-элементы. Потребовать, чтобы взамен применялся XML-атрибут, можно следующим образом:

```
[XmlAttribute] public int Age;
```

Именем элемента или атрибута можно управлять:

```
public class Person
{
  [XmlElement ("FirstName")] public string Name;
  [XmlAttribute ("RoughAge")] public int Age;
}
```

Ниже показан результат:

```
<Person RoughAge="30" ...>
  <FirstName>Stacey</FirstName>
</Person>
```

Стандартное пространство имен XML является пустым (в отличие от сериализатора на основе контрактов данных, который использует пространство имен типа). Для указания пространства имен XML атрибуты `[XmlElement]` и `[XmlAttribute]` поддерживают аргумент `Namespace`. Можно также назначить имя и пространство имен самому типу с помощью атрибута `[XmlRoot]`:

```
[XmlRoot ("Candidate", Namespace = "http://mynamespace/test/")]
public class Person { ... }
```

Здесь элементу `person` назначается имя `Candidate` и вдобавок устанавливается пространство имен для него и его дочерних элементов.

### Порядок следования XML-элементов

Класс `XmlSerializer` записывает элементы в порядке, в котором они определены в классе. Такое поведение можно изменить, указывая значение для аргумента `Order` в атрибуте `XmlElement`:

```
public class Person
{
  [XmlElement (Order = 2)] public string Name;
  [XmlElement (Order = 1)] public int Age;
}
```

Если аргумент `Order` вообще применяется, тогда он должен присутствовать везде.

Десериализатор не беспокоится по поводу порядка следования элементов – они могут появляться в любой последовательности и тип будет корректно десериализирован.

## Подклассы и дочерние объекты

### Создание подклассов из корневого типа

Предположим, что корневой тип имеет два подкласса:

```
public class Person { public string Name; }

public class Student : Person { }
public class Teacher : Person { }
```

и реализован многократно используемый метод для сериализации корневого типа:

```
public void SerializePerson (Person p, string path)
{
  XmlSerializer xs = new XmlSerializer (typeof (Person));
  using (Stream s = File.Create (path))
    xs.Serialize (s, p);
}
```

Чтобы данный метод работал с объектом `Student` или `Teacher`, экземпляр `XmlSerializer` должен быть информирован о существовании упомянутых подклассов. Сделать это можно двумя способами. Первый из них – зарегистрировать каждый подкласс с помощью атрибута `XmlInclude`:

```
[XmlInclude (typeof (Student))]
[XmlInclude (typeof (Teacher))]
public class Person { public string Name; }
```

Второй способ – указать каждый подтип при конструировании экземпляра `XmlSerializer`:

```
XmlSerializer xs = new XmlSerializer (typeof (Person),
                   new Type[] { typeof (Student), typeof (Teacher) } );
```

В любом случае сериализатор реагирует помещением подтипа в атрибут `type` (в точности как сериализатор на основе контрактов данных):

```
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="Student">
  <Name>Stacey</Name>
</Person>
```

Затем десериализатор узнает из атрибута `type`, что нужно создать объект типа `Student`, а не `Person`.

Именем, находящимся в XML-атрибуте `type`, можно управлять, применяя к подклассу атрибут `[XmlType]`:

```
[XmlType ("Candidate")]
public class Student : Person { }
```

Вот результат:

```
<Person xmlns:xsi="..."
        xsi:type="Candidate">
```

## Сериализация дочерних объектов

Класс `XmlSerializer` автоматически рекурсивно обрабатывает объектные ссылки, такие как поле `HomeAddress` в `Person`:

```
public class Person
{
  public string Name;
  public Address HomeAddress = new Address();
}
public class Address { public string Street, PostCode; }
```

Ниже представлена демонстрация:

```
Person p = new Person(); p.Name = "Stacey";
p.HomeAddress.Street = "Odo St";
p.HomeAddress.PostCode = "6020";
```

А вот результирующий XML-код:

```
<Person ... >
  <Name>Stacey</Name>
  <HomeAddress>
    <Street>Odo St</Street>
    <PostCode>6020</PostCode>
  </HomeAddress>
</Person>
```

При наличии двух полей или свойств, которые ссылаются на тот же самый объект, этот объект сериализируется дважды. Если эквивалентность ссылок должна быть предохранена, тогда придется использовать другой механизм сериализации.

## Создание подклассов из дочерних объектов

Предположим, что нужно сериализировать класс `Person`, который может ссылаться на *подклассы* `Address` следующим образом:

```
public class Address { public string Street, PostCode; }
public class USAddress : Address {  }
public class AUAddress : Address {  }

public class Person
{
  public string Name;
  public Address HomeAddress = new USAddress();
}
```

В зависимости от того, как должен структурироваться XML, решить задачу можно двумя отличающимися способами. Если требуется, чтобы имя элемента всегда соответствовало имени поля или свойства с подтипом, записанным в атрибуте `type`:

```
<Person ...>
  ...
  <HomeAddress xsi:type="USAddress">
    ...
  </HomeAddress>
</Person>
```

то необходимо применять атрибут `[XmlInclude]` для регистрации каждого подкласса с классом `Address`:

```
[XmlInclude (typeof (AUAddress))]
[XmlInclude (typeof (USAddress))]
public class Address
{
  public string Street, PostCode;
}
```

С другой стороны, если нужно, чтобы имя элемента отражало имя подтипа, давая примерно такой результат:

```
<Person ...>
  ...
  <USAddress>
    ...
  </USAddress>
</Person>
```

тогда взамен понадобится указывать множество атрибутов [XmlElement] для поля или свойства родительского типа:

```
public class Person
{
  public string Name;
  [XmlElement ("Address", typeof (Address))]
  [XmlElement ("AUAddress", typeof (AUAddress))]
  [XmlElement ("USAddress", typeof (USAddress))]
  public Address HomeAddress = new USAddress();
}
```

Каждый атрибут [XmlElement] сопоставляет имя элемента с типом. В случае принятия такого подхода атрибуты [XmlInclude] для типа Address не потребуются (хотя их наличие сериализацию не нарушит).

Если опустить имя элемента в [XmlElement] (и указать только тип), то будет использоваться стандартное имя типа (на которое оказывает влияние атрибут [XmlType], но не [XmlRoot]).

## Сериализация коллекций

Класс XmlSerializer распознает и сериализирует конкретные типы коллекций, не требуя какого-то вмешательства:

```
public class Person
{
  public string Name;
  public List<Address> Addresses = new List<Address>();
}
public class Address { public string Street, PostCode; }
```

Результирующий XML-код выглядит следующим образом:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    ...
  </Addresses>
</Person>
```

Атрибут [XmlArray] позволяет переименовывать *внешний* элемент (т.е. Addresses).

Атрибут [XmlArrayItem] позволяет переименовывать *внутренние* элементы (т.е. элементы Address).

Например, показанный далее класс:

```
public class Person
{
  public string Name;
  [XmlArray ("PreviousAddresses")]
  [XmlArrayItem ("Location")]
  public List<Address> Addresses = new List<Address>();
}
```

сериализируется так:

```
<Person ... >
  <Name>...</Name>
  <PreviousAddresses>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    ...
  </PreviousAddresses>
</Person>
```

Атрибуты `XmlArray` и `XmlArrayItem` также позволяют указывать пространства имен XML.

Для сериализации коллекций *без* внешнего элемента, например:

```
<Person ... >
  <Name>...</Name>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
</Person>
```

атрибут `[XmlElement]` необходимо добавить к полю или свойству коллекции:

```
public class Person
{
  ...
  [XmlElement ("Address")]
  public List<Address> Addresses = new List<Address>();
}
```

## Работа с элементами коллекции, являющимися подклассами

Правила для элементов коллекции, являющихся подклассами, естественным образом следуют из других правил, применяемых к подклассам. Чтобы закодировать элементы, являющиеся подклассами, с помощью атрибута `type`, например:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address xsi:type="AUAddress">
    ...
```

понадобится добавить атрибуты [XmlInclude] к базовому типу (Address), как делалось ранее. Это работает независимо от того, подавляется сериализация внешнего элемента или нет. Если элементы, являющиеся подклассами, должны именоваться в соответствии со своими типами, например:

```
<Person ... >
  <Name>...</Name>
  <!--начало необязательного внешнего элемента-->
  <AUAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </AUAddress>
  <USAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </USAddress>
  <!--конец необязательного внешнего элемента-->
</Person>
```

то для поля или свойства коллекции потребуется задать множество атрибутов [XmlArrayItem] или [XmlElement].

Указывайте множество атрибутов [XmlArrayItem], если нужно *включить* внешний элемент коллекции:

```
[XmlArrayItem ("Address",   typeof (Address))]
[XmlArrayItem ("AUAddress", typeof (AUAddress))]
[XmlArrayItem ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

Указывайте множество атрибутов [XmlElement], если нужно *исключить* внешний элемент коллекции:

```
[XmlElement ("Address",   typeof (Address))]
[XmlElement ("AUAddress", typeof (AUAddress))]
[XmlElement ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

## IXmlSerializable

Хотя сериализация XML на основе атрибутов обладает гибкостью, с ней связаны ограничения. Например, невозможно добавлять ловушки сериализации, равно как и сериализировать неоткрытые члены. Ее также неудобно использовать, если XML может представлять один и тот же элемент или атрибут несколькими разными путями.

Что касается последней проблемы, то границы можно несколько расширить, передавая конструктору XmlSerializer объект XmlAttributeOverrides. Тем не менее, наступает момент, когда проще принять императивный подход. За такую работу отвечает интерфейс IXmlSerializable:

```
public interface IXmlSerializable
{
  XmlSchema GetSchema();
  void ReadXml (XmlReader reader);
  void WriteXml (XmlWriter writer);
}
```

Реализация интерфейса IXmlSerializable обеспечивает полный контроль над читаемым или записываемым XML-кодом.

Класс коллекции, который реализует интерфейс `IXmlSerializable`, пропускает правила `XmlSerializer`, принятые для сериализации коллекций. Это может быть удобно, когда необходимо сериализировать коллекции с полезной нагрузкой – другими словами, с дополнительными полями или свойствами, которые иначе были бы проигнорированы.

Ниже описаны правила реализации интерфейса `IXmlSerializable`.

- Метод `ReadXml` должен читать внешний начальный элемент, затем содержимое и, наконец, внешний конечный элемент.
- Метод `WriteXml` должен записывать только содержимое.

Рассмотрим пример:

```
using System;
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;

public class Address : IXmlSerializable
{
  public string Street, PostCode;

  public XmlSchema GetSchema() { return null; }

  public void ReadXml(XmlReader reader)
  {
    reader.ReadStartElement();
    Street   = reader.ReadElementContentAsString ("Street", "");
    PostCode = reader.ReadElementContentAsString ("PostCode", "");
    reader.ReadEndElement();
  }

  public void WriteXml (XmlWriter writer)
  {
    writer.WriteElementString ("Street", Street);
    writer.WriteElementString ("PostCode", PostCode);
  }
}
```

Сериализация и десериализация экземпляра `Address` через `XmlSerializer` приводит к автоматическому вызову методов `WriteXml` и `ReadXml`. Более того, если класс `Person` определен следующим образом:

```
public class Person
{
  public string Name;
  public Address HomeAddress;
}
```

тогда обращение к реализации `IXmlSerializable` будет осуществляться выборочно для сериализации поля `HomeAddress`.

Классы `XmlReader` и `XmlWriter` подробно рассматривались в начале главы 11. Кроме того, в разделе "Шаблоны для использования `XmlReader/XmlWriter`" упомянутой главы были представлены примеры классов, совместимых с `IXmlSerializable`.

# 18

# Сборки

Сборка – это базовая единица развертывания в .NET, а также контейнер для всех типов. Сборка содержит скомпилированные типы с их кодом на промежуточном языке (Intermediate Language – IL), ресурсы времени выполнения и информацию, которая содействует ведению версий, безопасности и ссылкам на другие сборки. Сборка также определяет границы для распознавания типов и применения прав доступа. В общем случае сборка представляет собой одиночный файл Windows, называемый *переносимым исполняемым* (Portable Executable – PE) файлом, который имеет расширение `.exe` в случае приложения или `.dll` в случае многократно используемой библиотеки. Библиотека WinRT имеет расширение `.winmd` и подобна `.dll` за исключением того, что содержит только метаданные, но не код IL.

Большинство типов в главе находятся в следующих пространствах имен:

```
System.Reflection
System.Resources
System.Globalization
```

## Содержимое сборки

Сборка содержит четыре вида информации.

- *Манифест сборки.* Предоставляет сведения для исполняющей среды .NET, такие как имя сборки, версия, требуемые разрешения и ссылки на другие сборки.
- *Манифест приложения.* Предоставляет сведения для операционной системы (ОС), такие как способ развертывания сборки и необходимость в подъеме полномочий до административных.
- *Скомпилированные типы.* Скомпилированный код IL и метаданные типов, определенных внутри сборки.
- *Ресурсы.* Другие встроенные в сборку данные, такие как изображения и локализуемый текст.

Из всего перечисленного обязательным является только *манифест сборки*, хотя сборка почти всегда содержит скомпилированные типы (если только это не ссылочная сборка WinRT).

Сборки структурируются похожим образом, будь они исполняемыми файлами или библиотеками. Главное отличие исполняемого файла связано с тем, что в нем определена точка входа.

## Манифест сборки

Манифест сборки служит двум целям:

- описывает сборку для управляемой среды размещения;
- действует в качестве каталога для модулей, типов и ресурсов в сборке.

Таким образом, сборки являются *самоописательными*. Потребитель может обнаруживать данные, типы и функции всех сборок без необходимости в наличии дополнительных файлов.

Манифест сборки – не сущность, добавляемая к сборке явно; он встраивается в сборку автоматически во время компиляции.

Ниже представлены краткие сведения по функционально значащим данным, хранящимся в манифесте:

- простое имя сборки;
- номер версии (`AssemblyVersion`);
- открытый ключ и подписанный хеш сборки, если она имеет строгое имя;
- список ссылаемых сборок, включающий их версии и открытые ключи;
- список модулей, которые образуют сборку;
- список типов, определенных в сборке, и модулей, содержащих каждый тип;
- необязательный набор разрешений безопасности, запрашиваемых или отклоняемых сборкой (`SecurityPermission`);
- целевая культура в случае подчиненной сборки (`AssemblyCulture`).

Манифест также может хранить следующие информационные данные:

- полный заголовок и описание (`AssemblyTitle` и `AssemblyDescription`);
- информация о компании и авторском праве (`AssemblyCompany` и `AssemblyCopyright`);
- отображаемая версия (`AssemblyInformationalVersion`);
- дополнительные атрибуты для специальных данных.

Некоторые из перечисленных данных выводятся из аргументов, переданных компилятору, например, список ссылаемых сборок или открытый ключ для подписания сборки. Остальные данные поступают из атрибутов сборки, указанных в круглых скобках.

Просмотреть содержимое манифеста сборки можно с помощью инструмента .NET под названием `ildasm.exe`. В главе 19 будет показано, как делать то же самое программно с применением рефлексии.

### Указание атрибутов сборки

Большинством содержимого манифеста можно управлять с помощью атрибутов сборки. Например:

```
[assembly: AssemblyCopyright ("\x00a9 Corp Ltd. All rights reserved.")]
[assembly: AssemblyVersion ("2.3.2.1")]
```

Все объявления такого рода обычно определяются в одном файле внутри проекта. Для этой цели среда Visual Studio автоматически создает файл по имени `AssemblyInfo.cs` в папке `Properties` каждого нового проекта C# и предварительно заполняет его стандартным набором атрибутов сборки, которые предоставляют отправную точку для дальнейшей настройки.

## Манифест приложения

Манифест приложения – это XML-файл, который сообщает операционной системе информацию о сборке. Если манифест приложения предусмотрен, то он читается и обрабатывается перед тем, как среда размещения, управляемая .NET, загружает сборку, и может повлиять на способ запуска процесса приложения со стороны ОС.

Манифест приложения .NET имеет корневой элемент под названием `assembly` в пространстве имен XML вида `urn:schemas-microsoft-com:asm.v1`:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <!-- содержимое манифеста -->
</assembly>
```

Следующий манифест инструктирует ОС о запрашивании поднятия полномочий до административных:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

Последствия от требований поднятия полномочий до административных будут описаны в главе 21.

Приложения UWP имеют гораздо более сложный манифест, описанный в файле `Package.appxmanifest`. Он включает объявление функциональных возможностей программы, которые определяют разрешения, выдаваемые ОС. Простейший способ редактирования данного файла предусматривает использование среды Visual Studio, которая предлагает пользовательский интерфейс, доступный по двойному щелчку на файле манифеста.

### Развертывание манифеста приложения .NET

Развернуть манифест приложения .NET можно двумя способами:

- как файл со специальным именем, расположенный в той же папке, что и сборка;
- как встроенный внутрь самой сборки.

Имя отдельного файла манифеста должно совпадать с именем файла сборки и дополняться расширением `.manifest`. Таким образом, если сборка имеет имя `MyApp.exe`, то ее манифест должен называться `MyApp.exe.manifest`.

Чтобы встроить файл манифеста приложения в сборку, сначала нужно скомпилировать сборку, а затем применить инструмент .NET по имени `mt`, как показано ниже:

```
mt -manifest MyApp.exe.manifest -outputresource:MyApp.exe;#1
```

Инструмент .NET под названием `ildasm.exe` не замечает присутствия встроенного манифеста приложения. Тем не менее, среда Visual Studio указывает на наличие встроенного манифеста приложения, если дважды щелкнуть на сборке в проводнике решения (Solution Explorer).

## Модули

Содержимое сборки в действительности упаковано внутри одного или нескольких промежуточных контейнеров, которые называются *модулями*. Модуль соответствует файлу, хранящему содержимое сборки. Причина наличия такого дополнительного контейнерного уровня – позволить сборке охватывать множество файлов; эта возможность полезна при построении сборки, которая содержит скомпилированный код, написанный на смеси языков программирования.

На рис. 18.1 показан обычный случай сборки с единственным модулем, а на рис. 18.2 – случай многофайловой сборки. В многофайловой сборке "главный" модуль всегда содержит манифест; дополнительные модули могут содержать код IL и/или ресурсы. Манифест описывает относительное местоположение всех других модулей, формирующих сборку.

Многофайловые сборки должны компилироваться в командной строке: их поддержка в Visual Studio отсутствует. Для этого необходимо запустить компилятор `csc` с переключателем `/t`, чтобы создать каждый модуль, после чего скомпоновать модули посредством инструмента компоновки `al.exe`.

Хотя потребность в многофайловых сборках возникает редко, временами нужно принимать во внимание дополнительный контейнерный уровень, предлагаемый модулями – даже при работе только с одномодульными сборками. Основной такой сценарий связан с рефлексией (как показано в разделах "Рефлексия сборок" и "Выпуск сборок и типов" главы 19).

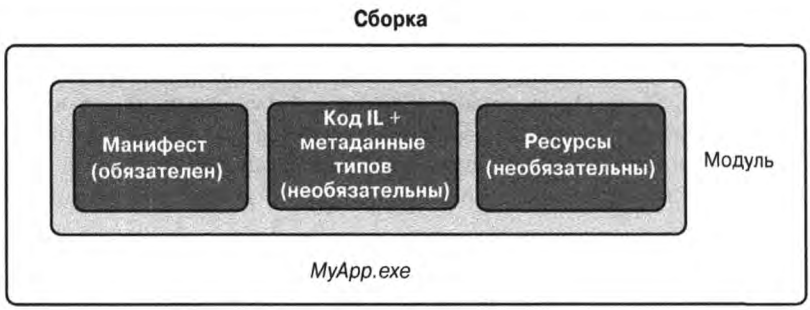



***Рис. 18.1.*** *Однофайловая сборка*

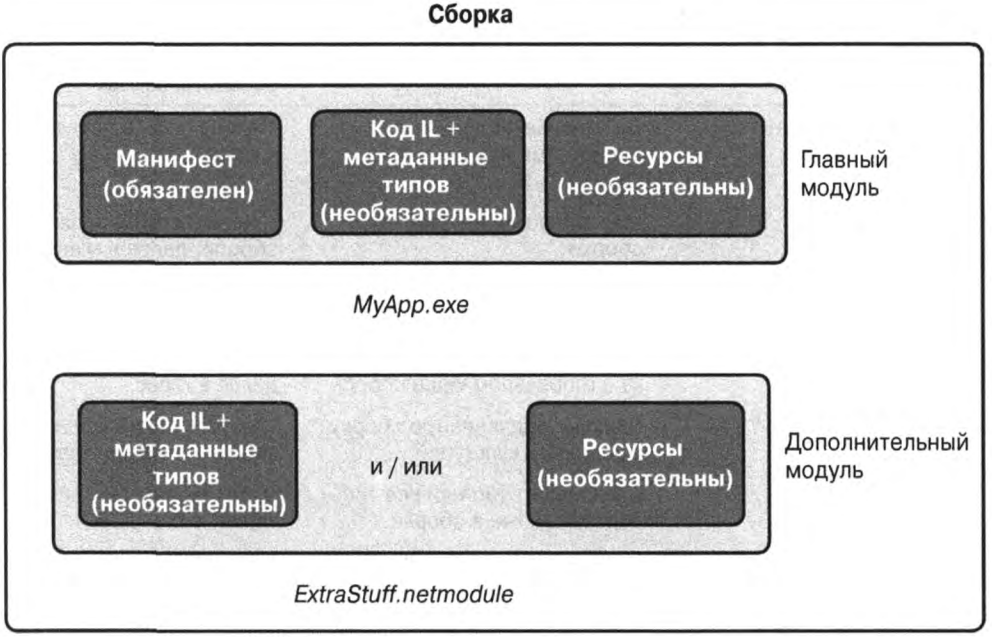



***Рис. 18.2.** Многофайловая сборка*

## Класс `Assembly`

Класс `Assembly` из пространства имен `System.Reflection` представляет собой шлюз для доступа к метаданным сборки во время выполнения. Существует несколько способов получения объекта сборки: простейший из них предусматривает использование свойства `Assembly` класса `Type`:

```
Assembly a = typeof (Program).Assembly;
```

или в приложениях UWP:

```
Assembly a = typeof (Program).GetTypeInfo().Assembly;
```

В настольных приложениях объект `Assembly` можно также получить вызовом одного из перечисленных ниже статических методов класса `Assembly`.

- `GetExecutingAssembly`. Возвращает сборку типа, в котором определена текущая выполняемая функция.
- `GetCallingAssembly`. Делает то же, что и метод `GetExecutingAssembly`, но для функции, которая вызвала текущую выполняемую функцию.
- `GetEntryAssembly`. Возвращает сборку, определяющую первоначальный метод точки входа в приложение.

После получения объекта `Assembly` можно применять его свойства и методы для запрашивания метаданных сборки и рефлексии ее типов. В табл. 18.1 представлена сводка по членам класса `Assembly`.

**Таблица 18.1. Члены класса `Assembly`**

| Члены | Назначение | Разделы, в которых рассматриваются |
|---|---|---|
| `FullName`, `GetName` | Возвращает полностью заданное имя или объект `AssemblyName` | "Имена сборок" далее в главе |
| `CodeBase`, `Location` | Местоположение файла сборки | "Распознавание и загрузка сборок" далее в главе |
| `Load`, `LoadFrom`, `LoadFile` | Вручную загружает сборку в текущий домен приложения | "Распознавание и загрузка сборок" далее в главе |
| `GlobalAssemblyCache` | Указывает, находится ли сборка в глобальном кеше сборок | "Глобальный кеш сборок" далее в главе |
| `GetSatelliteAssembly` | Находит подчиненную сборку с заданной культурой | "Ресурсы и подчиненные сборки" далее в главе |
| `GetType`, `GetTypes` | Возвращает тип или все типы, определенные в сборке | "Рефлексия и активизация типов" в главе 19 |
| `EntryPoint` | Возвращает метод точки входа в приложение как объект `MethodInfo` | "Рефлексия и вызов членов" в главе 19 |
| `GetModules`, `ManifestModule` | Возвращает все модули или главный модуль сборки | "Рефлексия сборок" в главе 19 |
| `GetCustomAttributes` | Возвращает атрибуты сборки | "Работа с атрибутами" в главе 19 |

# Строгие имена и подписание сборок

*Строго именованная* сборка имеет уникальное удостоверение, подделать которое невозможно. Оно получается за счет добавления к манифесту следующих метаданных:

- *уникального номера*, который принадлежит авторам сборки;
- *подписанного хеша* сборки, подтверждающего тот факт, что сборка создана владельцем уникального номера.

Здесь требуется пара открытого и секретного ключей. *Открытый ключ* предоставляет уникальный идентифицирующий номер, а *секретный ключ* облегчает подписание.

Подписание с помощью *строгого имени* – не то же самое, что подписание *Authenticode*. Система Authenticode рассматривается далее в главе.

Открытый ключ ценен для гарантирования уникальности ссылок на сборку: строго именованная сборка содержит в своем удостоверении открытый ключ. Подписание полезно для обеспечения безопасности – оно предотвращает подделку сборки злоумышленниками. Без вашего секретного ключа никто не сможет выпустить модифицированную версию сборки, не нарушив подпись (что вызовет ошибку во время загрузки сборки). Разумеется, кто-то мог бы подписать сборку с помощью другой пары

ключей, но это изменит удостоверение сборки. Любое приложение, ссылающееся на первоначальную сборку, будет избегать загрузки подделанной сборки, потому что маркеры открытого ключа записываются в ссылки.

Добавление строгого имени к сборке, ранее имеющей "слабое" имя, изменяет ее удостоверение. По указанной причине производственным сборкам имеет смысл назначать строгие имена с самого начала.

Строго именованная сборка также может быть зарегистрирована в глобальном кеше сборок.

## Назначение сборке строгого имени

Чтобы назначить сборке строгое имя, сначала понадобится с помощью утилиты `sn.exe` сгенерировать пару открытого и секретного ключей:

```
sn.exe -k MyKeyPair.snk
```

Утилита создаст новую пару ключей и сохранит ее в файле `MyApp.snk`. Если вы впоследствии потеряете этот файл, то навсегда утратите возможность перекомпиляции своей сборки с тем же самым удостоверением.

Затем сборку необходимо скомпилировать с переключателем `/keyfile`:

```
csc.exe /keyfile:MyKeyPair.snk Program.cs
```

Среда Visual Studio помогает выполнить оба шага посредством окна свойств проекта.

Строго именованная сборка не может ссылаться на слабо именованную сборку. Это еще одна веская причина назначать строгие имена всем производственным сборкам.

Одной и той же парой ключей можно подписывать множество сборок — если их простые имена отличаются, то они получат разные удостоверения. Выбор количества файлов с парами ключей внутри организации зависит от нескольких факторов. Наличие отдельной пары ключей для каждой сборки полезно, если в будущем планируется передача права владения конкретным приложением (вместе со сборками, на которые оно ссылается), поскольку в такой ситуации требуется минимальное обнародование внутренней информации. Однако тогда затрудняется создание политики безопасности, которая опознавала бы все ваши сборки. Кроме того, усложняется проверка достоверности динамически загружаемых сборок.

До выхода версии C# 2.0 компилятор не поддерживал переключатель `/keyfile`, и файл ключей нужно было указывать с помощью атрибута `AssemblyKeyFile`. В результате возникал риск в отношении безопасности, т.к. путь к файлу ключей оставался встроенным в метаданные сборки. Например, посредством `ildasm` довольно легко выяснить, что путь к файлу ключей, который использовался при подписании сборки `mscorlib` в CLR 1.1, выглядит следующим образом:

```
F:\qfe\Tools\devdiv\EcmaPublicKey.snk
```

Понятно, что для извлечения выгоды из такой информации необходим доступ к этой папке на машине сборки .NET Framework в Microsoft!

## Отложенное подписание

В организации с сотнями разработчиков может понадобиться ограничить доступ к парам ключей, применяемым для подписания сборок, по двум причинам:

- если случится утечка пары ключей, тогда ваши сборки больше не будут защищены от подделки;
- если произойдет утечка подписанной тестовой сборки, то злоумышленники смогут выдать ее за реальную сборку.

Тем не менее, сокрытие пар ключей от разработчиков означает невозможность компиляции и тестирования ими сборок с корректными удостоверениями. *Отложенное подписание* является системой, позволяющей обойти такую проблему.

Сборка с отложенной подписью помечается корректным открытым ключом, но не *подписывается* посредством секретного ключа. Сборка с отложенной подписью эквивалентна поддельной сборке и обычно отклоняется средой CLR. Однако разработчик инструктирует CLR о необходимости пропускать проверку достоверности сборок с отложенной подписью на *данном компьютере*, позволяя неподписанным сборкам запускаться. Когда наступает время для окончательного развертывания, владелец секретного ключа подписывает сборку с помощью действительной пары ключей.

Для отложенного подписания необходим файл, содержащий *только* открытый ключ. Его можно извлечь из пары ключей, запустив утилиту `sn` с переключателем `-p`:

```
sn -k KeyPair.snk
sn -p KeyPair.snk PublicKeyOnly.pk
```

Файл `KeyPair.snk` останется защищенным, а `PublicKeyOnly.pk` можно свободно распространять.

Получить `PublicKeyOnly.pk` можно также из существующей подписанной сборки, указав переключатель `-e`:

```
sn -e YourLibrary.dll PublicKeyOnly.pk
```

Далее можно произвести отложенное подписание с использованием `PublicKeyOnly.pk`, запустив компилятор `csc` с переключателем `/delaysign+`:

```
csc /delaysign+ /keyfile: PublicKeyOnly.pk /target:library YourLibrary.cs
```

Среда Visual Studio делает то же самое, если в окне свойств проекта отмечен флажок Delay sign (Отложенное подписание).

На следующем шаге исполняющей среде .NET сообщается, что она должна пропускать проверку удостоверений сборок на компьютерах разработки, где выполняются сборки с отложенным подписанием. Это можно делать на основе либо сборок, либо открытых ключей, запуская утилиту `sn` с переключателем `Vr`:

```
sn -Vr YourLibrary.dll
```

Среда Visual Studio не выполняет такой шаг автоматически. Проверку достоверности сборок потребуется отключить вручную в командной строке, иначе сборка с отложенной подписью не запустится.

Последний шаг предусматривает полное подписание сборки перед развертыванием. Именно на данном этапе пустая подпись заменяется реальной подписью, которая

может быть сгенерирована только при доступе к секретному ключу. Для этого нужно запустить утилиту sn с переключателем R:

```
sn -R YourLibrary.dll KeyPair.snk
```

Затем на машинах разработки можно восстановить проверку достоверности сборок:

```
sn -Vu YourLibrary.dll
```

Повторная компиляция приложений, ссылающихся на сборку с отложенной подписью, не требуется, т.к. изменилась только подпись сборки, но не ее *удостоверение*.

# Имена сборок

"Удостоверение" сборки содержит в себе четыре фрагмента метаданных из манифеста сборки:

- простое имя;
- версия ("0.0.0.0", если не указана);
- культура ("нейтральная", если сборка не является подчиненной);
- маркер открытого ключа ("пустой", если строгое имя не задано).

Простое имя поступает не из какого-то атрибута, а представляет собой имя файла, в который сборка была первоначально скомпилирована (не включая расширение). Таким образом, простое имя сборки System.Xml.dll выглядит как System.Xml. Переименование файла не изменяет простое имя сборки.

Номер версии берется из атрибута AssemblyVersion. Это строка, разделенная на четыре части:

*старший_номер.младший_номер.компоновка.редакция*

Указать номер версии можно следующим образом:

```
[assembly: AssemblyVersion ("2.5.6.7")]
```

Культура поступает из атрибута AssemblyCulture и применяется к подчиненным сборкам, которые описаны в разделе "Ресурсы и подчиненные сборки" далее в главе.

Маркер открытого ключа извлекается из пары ключей, предоставляемых на этапе компиляции через переключатель /keyfile, как было показано выше в разделе "Назначение сборке строгого имени".

## Полностью заданные имена

Полностью заданное имя сборки — это строка, включающая все четыре идентифицирующих компонента в таком формате:

*простое_имя*, Version=*версия*, Culture=*культура*, PublicKeyToken=*открытый_ключ*

Например, вот полностью заданное имя сборки System.Xml.dll:

```
"System.Xml, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
```

Если сборка не имеет атрибута AssemblyVersion, то ее версией будет 0.0.0.0. Для неподписанной сборки маркер открытого ключа пуст.

Свойство FullName объекта Assembly возвращает полностью заданное имя сборки. При занесении ссылок на сборки в манифест компилятор всегда использует полностью заданные имена.

Полностью заданное имя не включает путь к каталогу, чтобы тем самым содействовать в нахождении сборки на диске. Поиск сборки, расположенной в другом каталоге, является совершенно другой темой, которой посвящен раздел "Распознавание и загрузка сборок" далее в главе.

## Класс `AssemblyName`

`AssemblyName` – класс с типизированными свойствами для каждого из четырех компонентов полностью заданного имени сборки. Класс `AssemblyName` служит двум целям:

- он разбирает или строит полностью заданное имя сборки;
- он хранит некоторые дополнительные данные, помогающие распознавать (находить) сборку.

Получить объект `AssemblyName` можно любым из следующих способов:

- создать объект `AssemblyName`, предоставив полностью заданное имя;
- вызвать метод `GetName` на существующем объекте `Assembly`;
- вызвать метод `AssemblyName.GetAssemblyName`, указав путь к файлу сборки на диске (только для настольных приложений).

Объект `AssemblyName` можно также создать безо всяких аргументов и затем установить все его свойства, построив в итоге полностью заданное имя. Когда объект `AssemblyName` сконструирован в подобной манере, он является изменяемым.

Ниже перечислены важные свойства и методы `AssemblyName`:

```
string      FullName      { get; }         // Полностью заданное имя
string      Name          { get; set; }    // Простое имя
Version     Version       { get; set; }    // Версия сборки
CultureInfo CultureInfo   { get; set; }    // Для подчиненных сборок
string      CodeBase      { get; set; }    // Местоположение

byte[]      GetPublicKey();                // 160 байтов
void        SetPublicKey (byte[] key);
byte[]      GetPublicKeyToken();           // 8-байтовая версия
void        SetPublicKeyToken (byte[] publicKeyToken);
```

Само свойство `Version` – это строго типизированное представление со свойствами `Major`, `Minor`, `Build` и `Revision`. Метод `GetPublicKey` возвращает криптографически стойкий открытый ключ; метод `GetPublicKeyToken` возвращает последние восемь байтов, применяемых в устанавливаемом удостоверении.

Вот как использовать `AssemblyName` для получения простого имени сборки:

```
Console.WriteLine (typeof (string).Assembly.GetName().Name);  // mscorlib
```

А так извлекается версия сборки:

```
string v = myAssembly.GetName().Version.ToString();
```

Свойство `CodeBase` более подробно рассматривается в разделе "Распознавание и загрузка сборок" далее в главе.

## Информационная и файловая версии сборки

Поскольку версия является неотъемлемой частью имени сборки, изменение атрибута `AssemblyVersion` приводит к изменению удостоверения сборки. Это влияет на совместимость со ссылающимися сборками, что может оказаться нежелательным при

выполнении обновлений, не нарушающих работу. Для решения указанной проблемы предназначены два других независимых атрибута уровня сборки, которые позволяют выражать сведения, связанные с версией; оба атрибута среда CLR игнорирует.

- `AssemblyInformationalVersion`. Версия, отображаемая конечному пользователю. Она видна в поле Product Version (Версия продукта) диалогового окна свойств файла. Здесь можно указывать любую строку, например, "5.1 Beta 2". Обычно всем сборкам в приложении будет назначаться один и тот же номер информационной версии.
- `AssemblyFileVersion`. Позволяет ссылаться на номер компоновки для данной сборки. Такой номер отображается в поле File Version (Файловая версия) диалогового окна свойств файла. Как и `AssemblyVersion`, атрибут должен содержать строку, которая состоит максимум из четырех чисел, разделенных точками.

# Подпись Authenticode

*Authenticode* – это система подписания кода, назначение которой заключается в подтверждении удостоверения издателя. Система Authenticode и подписание с помощью *строгого имени* не зависят друг от друга: подписать сборку можно посредством либо какой-то одной, либо обеих систем.

В то время как подписание с помощью строгого имени подтверждает, что сборки A, B и C поступают от одного и того же издателя (предполагая, что не произошла утечка секретного ключа), они не позволяют выяснить, кто конкретно этот издатель. Чтобы узнать, кто является издателем – Джо Албахари или компания Microsoft Corporation – нужна система Authenticode.

Система Authenticode полезна при загрузке программ из Интернета, т.к. она дает гарантию того, что программа поступает от издателя, зарегистрированного в центре сертификации (Certificate Authority), и не была по пути модифицирована. Данная система также обеспечивает выдачу предупреждения о неизвестном издателе (Unknown Publisher), показанного на рис. 18.3, когда загруженное приложение запускается впервые. Кроме того, подписание Authenticode обязательно при отправке приложений в магазин Windows Store и при построении сборок в общем как часть программы Windows Logo.

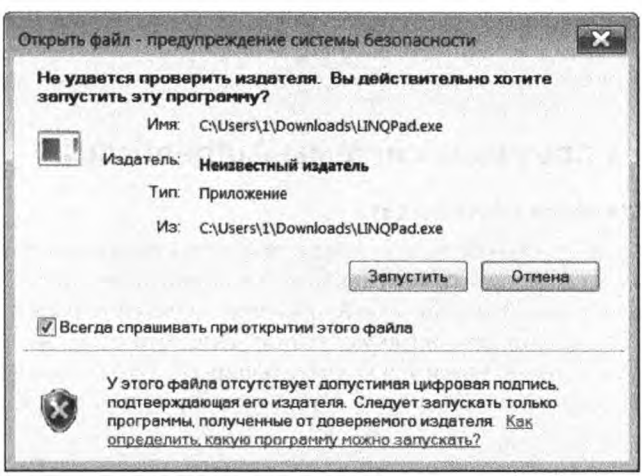



***Рис. 18.3. Предупреждение о неподписанном файле***

Система Authenticode работает не только со сборками .NET, но также с неуправляемыми исполняемыми и двоичными модулями, такими как элементы управления ActiveX или файлы развертывания `.msi`. Разумеется, система Authenticode не гарантирует, что программа свободна от вредоносного кода – хотя делает это менее вероятным. Дело в том, что физическое или юридическое лицо добровольно указало под исполняемым модулем или библиотекой свое реальное имя (подкрепленное паспортом или документом компании).

Среда CLR не трактует подпись Authenticode как часть удостоверения сборки. Тем не менее, как вскоре будет показано, она может читать и проверять достоверность подписей Authenticode по требованию.

Подписание с помощью Authenticode требует обращения в *центр сертификации* (Certificate Authority – CA) с доказательством вашей персональной личности или удостоверения компании (учредительный договор и т.п.). Как только в CA проверят предъявленные документы, они выдадут сертификат подписания кода X.509, который обычно действителен от одного до пяти лет. Сертификат позволяет подписывать сборки с применением утилиты `signtool`. Можно также создать сертификат самостоятельно с помощью утилиты `makecert`, однако он будет распознаваться только на компьютерах, на которых явно устанавливается.

Тот факт, что сертификаты (не подписанные самостоятельно) могут работать на любом компьютере, опирается на инфраструктуру открытых ключей. По существу ваш сертификат подписывается с помощью другого сертификата, принадлежащего CA. Центр сертификации является доверенным, потому что все CA загружаются в операционную систему (чтобы увидеть их, откройте панель управления Windows, выберите элемент Свойства браузера, в открывшемся диалоговом окне перейдите на вкладку Содержание, щелкните на кнопке Сертификаты и в диалоговом окне Сертификаты перейдите на вкладку Доверенные корневые центры сертификации). Центр сертификации может отозвать сертификат издателя, если произошла его утечка, поэтому проверка подписи Authenticode требует периодического запрашивания у CA актуальных списков отозванных сертификатов.

Из-за того, что система Authenticode использует криптографические подписи, подпись Authenticode становится недействительной, если кто-то впоследствии подделает файл. Вопросы, связанные с криптографией, хешированием и подписанием, рассматриваются в главе 21.

## Подписание с помощью системы Authenticode

### Получение и установка сертификата

Первый шаг связан с получением сертификата для подписания кода в CA (как объясняется во врезке "Где получать сертификат для подписания кода?"). Затем с сертификатом можно либо работать как с файлом, защищенным паролем, либо загрузить его в хранилище сертификатов на компьютере. Преимущество второго варианта в том, что при подписании не придется указывать пароль. Это позволяет избавиться от явного помещения пароля в сценарии построения сборок или пакетные файлы.

### Где получать сертификат для подписания кода?

В Windows предварительно загружается несколько корневых центров сертификации. В их число входят (в скобках указана цена за годовой сертификат для подписания кода на время публикации книги): Comodo ($180), Go Daddy ($249), GlobalSign ($400), DigiCert ($223), thawte ($299) и Symantec ($499).

Существует также торговый посредник Ksoftware (`http://www.ksoftware.net`), который в настоящее время предлагает сертификаты для подписания кода от Comodo за $84 в год.

Сертификаты Authenticode, выдаваемые Ksoftware, Comodo, Go Daddy и GlobalSign, рекламируются как менее ограничивающие в том, что с их помощью можно также подписывать программы для платформ, отличающихся от Microsoft. В остальном продукты от всех поставщиков функционально эквивалентны.

Обратите внимание, что сертификат для SSL в общем случае не может применяться для подписания с помощью Authenticode (несмотря на использование той же самой инфраструктуры X.509). Частично это обусловлено тем, что сертификат для SSL касается доказательства прав собственности на домен, а сертификат Authenticode связан с подтверждением личности.

Чтобы загрузить сертификат в хранилище сертификатов на компьютере, откройте панель управления Windows, выберите элемент Свойства браузера, в открывшемся диалоговом окне перейдите на вкладку Содержание, щелкните на кнопке Сертификаты и в диалоговом окне Сертификаты щелкните на кнопке Импорт. После того как мастер импорта сертификатов завершит работу, при выбранном сертификате щелкните на кнопке Просмотр, в открывшемся диалоговом окне Сертификат перейдите на вкладку Состав и скопируйте *отпечаток* сертификата. Это хеш SHA-1, который впоследствии понадобится для удостоверения сертификата во время подписания.

Если вы также хотите подписать свою сборку с помощью строгого имени (что настоятельно рекомендуется), то должны делать это *до* подписания посредством Authenticode. Причина в том, что среде CLR известно о подписях Authenticode, но не наоборот. Таким образом, если вы подпишете свою сборку с помощью строгого имени *после* ее подписания с применением Authenticode, то система Authenticode будет рассматривать добавление средой CLR строгого имени как неавторизованную модификацию и считать, что сборка подделана.

### Подписание с помощью `signtool.exe`

Подписывать свои программы с использованием системы Authenticode можно посредством утилиты `signtool`, входящей в состав Visual Studio. Когда утилита `signtool` запускается с флагом `signwizard`, она отображает пользовательский интерфейс; в противном случае командная строка для ее запуска выглядит следующим образом:

```
signtool sign /sha1 (отпечаток) имя_файла
```

Отпечаток – это то, что можно взять из хранилища сертификатов на компьютере. (Если сертификат находится в файле, тогда необходимо указать имя файла в переключателе `/f` и пароль в переключателе `/p`.)

Например:

```
signtool sign /sha1 ff813c473dc93aaca4bac681df472b037fa220b3 LINQPad.exe
```

Можно также задать описание и URL продукта в переключателях /d и /du:

```
... /d LINQPad /du http://www.linqpad.net
```

В большинстве случаев будет также указываться *сервер отметок времени.*

### Отметки времени

После истечения срока действия сертификата вы больше не сможете подписывать программы. Тем не менее, программы, которые были подписаны *до* истечения срока действия, по-прежнему будут действительными, если при подписании указывался *сервер отметок времени* с помощью переключателя /t. Для такой цели центр сертификации предоставляет URI: ниже показан URI для Comodo (или Ksoftware):

```
... /t http://timestamp.comodoca.com/authenticode
```

### Проверка, подписана ли программа

Простейший способ увидеть подпись Authenticode для файла – просмотреть свойства файла в проводнике Windows (на вкладке цифровых сертификатов). Утилита signtool также предоставляет такую возможность.

## Проверка достоверности подписей Authenticode

Проверять достоверность подписей Authenticode может как ОС, так и среда CLR.

Система Windows проверяет подписи Authenticode перед запуском программы, помеченной как "заблокированная", что на практике означает запуск программы в первый раз после ее загрузки из Интернета. Состояние информации Authenticode – или факт ее отсутствия – затем отображается в диалоговом окне, которое было показано ранее на рис. 18.3.

Среда CLR читает и проверяет подписи Authenticode при запрашивании удостоверения сборки. Вот как это делается:

```
Publisher p = некотораяСборка.Evidence.GetHostEvidence<Publisher>();
```

Класс Publisher (из пространства имен System.Security.Policy) открывает доступ к свойству Certificate. Если оно возвращает значение не null, то сборка подписана с помощью Authenticode. Затем данный объект можно запросить для получения сведений о сертификате.

До выхода версии .NET Framework 4.0 среда CLR читала и проверяла достоверность подписей Authenticode, когда сборка была загружена – вместо того, чтобы ожидать вызова метода GetHostEvidence. Результатом были потенциально пагубные последствия в отношении производительности, т.к. проверка Authenticode могла обращаться к центру сертификации для обновления списка отозванных сертификатов, что иногда занимало до 30 секунд (прежде чем отказать) в случае проблем с подключением к Интернету. По данной причине подписания посредством Authenticode сборок .NET 3.5 или предшествующих версий лучше по возможности избегать. (Хотя вполне нормально подписывать установочные файлы .msi.)

Если программа имеет недействительную или не позволяющую проверить подпись Authenticode, то вне зависимости от версии .NET Framework среда CLR просто делает такую информацию доступной через метод GetHostEvidence: она никогда не отображает предупреждение пользователю и не препятствует запуску сборки.

Как упоминалось ранее, подпись Authenticode не влияет на удостоверение или на *имя* сборки.

# Глобальный кеш сборок

В качестве части установки .NET Framework на компьютере создается центральный репозиторий для хранения сборок .NET, который называется *глобальным кешем сборок* (Global Assembly Cache – GAC). Глобальный кеш сборок содержит централизованную копию самой инфраструктуры .NET Framework, и в него также можно помещать собственные сборки.

Главным фактором в решении о том, загружать ли сборки в GAC, является поддержка версий. Для сборок в GAC поддержка версий централизуется на уровне машины и управляется администратором компьютера. Для сборок за пределами GAC поддержка версий производится на основе приложений, так что каждое приложение самостоятельно заботится о своих зависимостях и обновлениях (обычно сохраняя собственную копию каждой сборки, на которую имеется ссылка).

Глобальный кеш сборок полезен в меньшинстве случаев, когда централизованная поддержка версий на уровне машины по-настоящему выгодна. Например, пусть есть набор независимых подключаемых модулей, каждый из которых ссылается на ряд разделяемых сборок. Мы предполагаем, что каждый подключаемый модуль находится в собственном каталоге, и по этой причине существует вероятность наличия нескольких копий какой-то разделяемой сборки (возможно, более поздних версий). Далее представим, что ради эффективности или совместимости типов размещающее приложение желает выполнения загрузки каждой разделяемой сборки только один раз. Теперь задача распознавания сборок для размещающего приложения усложняется, требуя тщательного планирования и понимания тонкостей контекстов загрузки сборок. Простое решение в таком случае предусматривает помещение разделяемых сборок в GAC, тем самым гарантируя, что при распознавании сборок среда CLR всегда будет принимать простые и согласованные решения.

Тем не менее, в более типичных сценариях применения GAC лучше избегать, чтобы не столкнуться с перечисленными ниже последствиями.

- Развертывание XCOPY или ClickOnce перестает быть возможным; для установки приложения потребуются административные привилегии.
- Для обновления сборок в GAC также нужны административные привилегии.
- Использование GAC может усложнить разработку и тестирование, поскольку механизм распознавания сборок CLR (*Fusion*) всегда отдает предпочтение сборкам GAC перед локальными копиями.
- Поддержка версий и выполнение *бок о бок* требует определенного планирования, и какая-то ошибка может нарушить работу других приложений.

В качестве положительной стороны GAC может улучшить показатели времени запуска для очень крупных сборок, т.к. среда CLR проверяет подписи сборок в GAC только однажды при их установке туда, а не каждый раз, когда сборка загружается. В процентном отношении это имеет значение, если для сборок генерируются образы в машинном коде с помощью инструмента `ngen.exe` с выбором неперекрывающихся базовых адресов. Такие проблемы подробно описаны в онлайновой статье "To NGen or Not to NGen?" ("Использовать NGen или нет?") на сайте MSDN.

Сборки в GAC всегда являются полностью доверенными – даже когда они вызываются из сборки, выполняющейся в песочнице с ограниченными разрешениями. Мы обудим данный аспект более подробно в главе 21.

## Установка сборок в GAC

Первым шагом при установке сборки в GAC является назначение ей строгого имени. Затем сборку можно установить с помощью инструмента командной строки .NET под названием `gacutil`:

```
gacutil /i MyAssembly.dll
```

Если в GAC уже имеется сборка с *тем же самым открытым ключом и версией*, тогда она обновляется. Предварительно удалять старую сборку не требуется.

Вот как удалить сборку (обратите внимание, что расширение файла не указывается):

```
gacutil /u MyAssembly
```

Установку сборок в GAC можно также задать как часть проекта установки в Visual Studio. Запуск `gacutil` с переключателем `/l` позволяет получить список всех сборок в GAC.

После того, как сборка загружена в GAC, приложения могут ссылаться на нее, не нуждаясь в локальной копии данной сборки.

Если локальная копия *присутствует*, то она *игнорируется в пользу образа из GAC*. Это означает, что ссылаться или тестировать перекомпилированную версию сборки можно только после ее обновления в GAC. Сказанное остается справедливым при условии предохранения версии и удостоверения сборки.

## GAC и поддержка версий

Изменение атрибута сборки `AssemblyVersion` дает совершенно новое удостоверение. В целях иллюстрации представим, что вы написали сборку `utils`, присвоили ей версию 1.0.0.0, назначили строгое имя и затем установили ее в GAC. Предположим, что позже вы добавили в сборку несколько новых возможностей, изменили ее версию на 1.0.0.1, перекомпилировали сборку и переустановили ее в GAC. Вместо переписывания исходной сборки GAC теперь содержит сборки *обеих* версий. Вот что это означает:

- при компиляции другого приложения, применяющего сборку `utils`, можно выбирать, на какую версию ссылаться;
- любое приложение, ранее скомпилированное со ссылкой на сборку `utils` версии 1.0.0.0, *продолжит на нее ссылаться*.

Результат называется выполнением *бок о бок*. Выполнение бок о бок предотвращает проблему "ада DLL" (DLL hell), которая иначе могла бы случиться, когда разделяемая сборка обновляется односторонне: приложения, построенные для предыдущей версии сборки, могут неожиданно перестать работать.

Однако сложность возникает, когда требуется применить исправления ошибок или незначительные обновления к существующим сборкам. В такой ситуации на выбор есть два варианта:

- переустановить исправленную сборку в GAC с тем же самым номером версии;
- скомпилировать исправленную сборку с новым номером версии и установить ее в GAC.

Трудность первого варианта связана с невозможностью *избирательного* применения обновлений к определенным приложениям. Обновлять можно либо все приложения, либо ни одного. Трудность второго варианта заключается в том, что приложения не смогут нормально пользоваться более новой версией сборки без перекомпиляции. Существует обходной путь: создать *политику издателя*, разрешающую перенаправление версий сборки, но его ценой будет увеличенная сложность развертывания.

Выполнение бок о бок хорошо подходит для смягчения некоторых проблем, связанных с разделяемыми сборками. Если вы вообще откажетесь от применения GAC, взамен позволив каждому приложению поддерживать собственную копию сборки `utils`, то устраните *все* проблемы, связанные с разделяемыми сборками!

# Ресурсы и подчиненные сборки

Приложение обычно содержит не только исполняемый код, но и такие элементы, как текст, изображения или XML-файлы. Содержимое подобного рода может быть представлено в сборке посредством *ресурсов*. Для ресурсов предусмотрены два перекрывающихся сценария использования:

- встраивание данных, которые не могут располагаться в исходном коде, вроде изображений;
- хранение данных, которым может потребоваться перевод в многоязычном приложении.

Ресурс сборки в конечном итоге представляет собой байтовый поток с именем. О сборке можно говорить, что она содержит словарь байтовых массивов со строковыми ключами. Вот что можно увидеть с помощью утилиты `ildasm`, если дизассемблировать сборку, которая содержит ресурсы `banner.jpg` и `data.xml`:

```
.mresource public banner.jpg
{
  // Offset: 0x00000F58 Length: 0x000004F6
  // Смещение: 0x00000F58 Длина: 0x000004F6
}
.mresource public data.xml
{
  // Offset: 0x00001458 Length: 0x0000027E
  // Смещение: 0x00001458 Длина: 0x0000027E
}
```

В данном случае элементы `banner.jpg` и `data.xml` были включены прямо в сборку – каждый в виде собственного встроенного ресурса. Это простейший способ работы.

Инфраструктура .NET Framework также позволяет добавлять содержимое через промежуточные контейнеры `.resources`. Они предназначены для хранения содержимого, которое может требовать перевода на разные языки. Локализованные контейнеры `.resources` могут быть упакованы как отдельные подчиненные сборки, которые автоматически выбираются во время выполнения на основе языка операционной системы пользователя.

На рис. 18.4 показана сборка, содержащая два напрямую встроенных ресурса, а также контейнер `.resources` по имени `welcome.resources`, для которого созданы две локализованные подчиненные сборки.

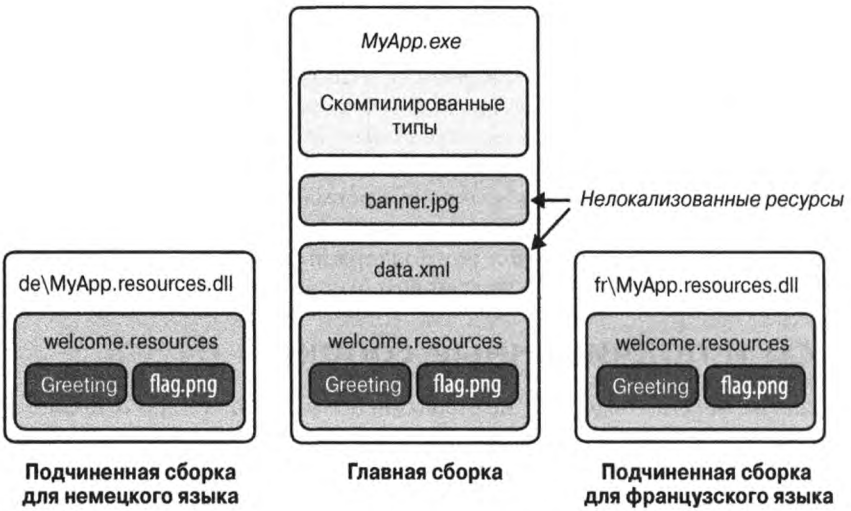



***Рис. 18.4.** Ресурсы*

## Встраивание ресурсов напрямую

Встраивание ресурсов внутрь сборок в приложениях Window Store не поддерживается. Взамен любые дополнительные файлы необходимо добавлять в пакет развертывания и обращаться к ним, читая в приложении объект `StorageFolder` (свойство `Package.Current.InstalledLocation`).

Чтобы встроить ресурс внутрь сборки напрямую в командной строке, необходимо использовать при компиляции переключатель `/resource`:

```
csc /resource:banner.jpg /resource:data.xml MyApp.cs
```

Дополнительно можно указать, что в сборке ресурс получит другое имя:

```
csc /resource:<имя-файла>,<имя-ресурса>
```

Чтобы встроить ресурс внутрь сборки напрямую с применением Visual Studio, понадобится выполнить следующие действия:

- добавить файл к проекту;
- установить действие построения проекта в Embedded Resource (Встроенный ресурс).

Среда Visual Studio всегда предваряет имена ресурсов стандартным пространством имен проекта, а также именами всех подпапок, ведущих к файлу. Таким образом, если стандартным пространством имен проекта было `Westwind.Reports`, а файл назывался `banner.jpg` и находился в папке `pictures`, то имя ресурса выглядело бы как `Westwind.Reports.pictures.banner.jpg`.

Имена ресурсов чувствительны к регистру символов, что делает имена подпапок с ресурсами в Visual Studio фактически чувствительными к регистру.

Чтобы извлечь ресурс, необходимо вызвать метод GetManifestResourceStream на объекте сборки, содержащей ресурс. Упомянутый метод возвращает поток, который затем допускается читать подобно любому другому потоку:

```
Assembly a = Assembly.GetEntryAssembly();

using (Stream s = a.GetManifestResourceStream ("TestProject.data.xml"))
using (XmlReader r = XmlReader.Create (s))
  ...
System.Drawing.Image image;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  image = System.Drawing.Image.FromStream (s);
```

Возвращенный поток поддерживает позиционирование, поэтому можно поступить и так:

```
byte[] data;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

Если для встраивания ресурса используется Visual Studio, тогда нужно не забыть о включении префикса, основанного на пространстве имен. Во избежание ошибки префикс разрешено указывать в отдельном аргументе с применением *типа*. В качестве префикса используется пространство имен данного типа:

```
using (Stream s = a.GetManifestResourceStream (typeof (X), "XmlData.xml"))
```

Здесь X может быть любым типом с желаемым пространством имен ресурса (обычно это тип в той же папке проекта).

Установка действия построения в Visual Studio для элемента проекта в Resource (Ресурс) внутри WPF-приложения — *не* то же, что установка его действия построения в Embedded Resource. В первом случае фактически добавляется элемент в файл .resources по имени *<ИмяСборки>*.g.resources, к содержимому которого можно получать доступ через класс Application инфраструктуры WPF, применяя URI в качестве ключа.

В инфраструктуре WPF вдобавок переопределен термин "ресурс", что еще больше увеличивает путаницу. *Статические ресурсы* и *динамические реурсы* не имеют никакого отношения к ресурсам сборки!

Метод GetManifestResourceNames возвращает имена всех ресурсов в сборке.

## Файлы .resources

Инфраструктура .NET Framework также позволяет добавлять файлы .resources, которые являются контейнерами для потенциально локализуемого содержимого. В итоге файл .resources становится встроенным ресурсом внутри сборки — точно так же, как файл другого вида. Разница заключается в следующем:

- первым делом содержимое должно быть упаковано в файл .resources;
- доступ к содержимому производится через объект ResourceManager или URI типа "pack", а не посредством метода GetManifestResourceStream.

Файлы .resources являются двоичными и не предназначены для редактирования человеком; таким образом, при работе с ними придется полагаться на инструменты, предоставляемые .NET Framework и Visual Studio.

Стандартный подход со строками или простыми типами данных предусматривает применение файла в формате `.resx`, который может быть преобразован в файл `.resources` с помощью Visual Studio либо инструмента `resgen`. Формат `.resx` также подходит для изображений, предназначенных для приложения Windows Forms или ASP.NET.

В приложении WPF должно использоваться действие построения Resource среды Visual Studio для изображений или похожего содержимого, нуждающегося в ссылке через URI. Это применимо вне зависимости от того, необходима локализация или нет.

Мы опишем упомянутые действия в последующих разделах.

## Файлы `.resx`

Инфраструктура .NET Framework позволяет добавлять файл `.resx`, имеющий формат этапа проектирования, который предназначен для получения файлов `.resources`. Файл `.resx` использует XML и структурирован в виде пар "имя/значение", как показано ниже:

```
<root>
  <data name="Greeting">
    <value>hello</value>
  </data>
  <data name="DefaultFontSize" type="System.Int32, mscorlib">
    <value>10</value>
  </data>
</root>
```

Чтобы создать файл `.resx` в Visual Studio, понадобится добавить элемент проекта типа Resources File (Файл ресурсов). Оставшаяся часть работы будет произведена автоматически:

- создается корректный заголовок;
- предоставляется визуальный конструктор для добавления строк, изображений, файлов и других видов данных;
- файл `.resx` автоматически преобразуется в формат `.resources` и встраивается в сборку при компиляции;
- генерируется класс, предназначенный для облегчения доступа к данным в более позднее время.

Визуальный конструктор ресурсов добавляет изображения как объекты типа `Image` (сборка `System.Drawing.dll`), а не как байтовые массивы, делая изображения неподходящими для приложений WPF.

### Создание файла `.resx` в командной строке

Если вы работаете в командной строке, то должны начинать с файла `.resx`, имеющего допустимый заголовок. Достичь этого проще всего, создав простой файл `.resx` программно. Такую работу делает класс `System.Resources.ResXResourceWriter` (который, как ни странно, находится в сборке `System.Windows.Forms.dll`):

```
using (ResXResourceWriter w = new ResXResourceWriter ("welcome.resx")) { }
```

Далее можно либо продолжить добавление ресурсов с помощью класса `ResXResourceWriter` (вызывая его метод `AddResource`), либо вручную отредактировать записанный файл `.resx`.

Простейший способ работы с изображениями предусматривает трактовку файлов изображений как двоичных данных и преобразование их содержимого в изображения во время извлечения. Описанное решение более универсально, чем кодирование изображений в виде объектов типа `Image`. В файл `.resx` можно помещать двоичные данные, представленные в формате Base64:

```
<data name="flag.png" type="System.Byte[], mscorlib">
  <value>Qk32BAAAAAAAAHYAAAAoAAAAMAMDAwACAgIAAAAD/AA....</value>
</data>
```

или указывать ссылку на другой файл, который затем читается инструментом `resgen`:

```
<data name="flag.png"
  type="System.Resources.ResXFileRef, System.Windows.Forms">
  <value>flag.png;System.Byte[], mscorlib</value>
</data>
```

По завершении файл `.resx` потребуется преобразовать с помощью `resgen`. Следующая команда выполняет преобразование файла `welcome.resx` в `welcome.resources`:

```
resgen welcome.resx
```

В качестве финального шага файл `.resources` включается в процесс компиляции:

```
csc /resources:welcome.resources MyApp.cs
```

### Чтение файлов `.resources`

В случае создания файла `.resx` в Visual Studio автоматически генерируется класс с таким же именем, который имеет свойства для извлечения каждого элемента.

Класс `ResourceManager` читает файлы `.resources`, встроенные внутрь сборки:

```
ResourceManager r = new ResourceManager ("welcome",
                                Assembly.GetExecutingAssembly());
```

(Если ресурс был скомпилирован в Visual Studio, тогда первый аргумент должен быть предварен названием пространства имен.)

Далее можно получить доступ к содержимому, вызывая метод `GetString` или `GetObject` и выполняя приведение:

```
string greeting = r.GetString ("Greeting");
int fontSize = (int) r.GetObject ("DefaultFontSize");
Image image = (Image) r.GetObject ("flag.png");     // (Среда Visual Studio)
byte[] imgData = (byte[]) r.GetObject ("flag.png"); // (Командная строка)
```

Перечисление содержимого файла `.resources` производится следующим образом:

```
ResourceManager r = new ResourceManager (...);
ResourceSet set = r.GetResourceSet (CultureInfo.CurrentUICulture,
                                    true, true);
foreach (System.Collections.DictionaryEntry entry in set)
  Console.WriteLine (entry.Key);
```

### Создание ресурса с доступом через URI типа "pack" в Visual Studio

В приложении WPF файлы XAML должны иметь возможность доступа к ресурсам по URI. Например:

```
<Button>
  <Image Height="50" Source="flag.png"/>
</Button>
```

Или, если ресурс находится в другой сборке:

```
<Button>
  <Image Height="50" Source="UtilsAssembly;Component/flag.png"/>
</Button>
```

(`Component` — литеральное ключевое слово.)

Для создания ресурсов, которые могут загружаться в подобной манере, применять файлы `.resx` не получится. Взамен придется добавлять файлы в проект и устанавливать для них действие построения в Resource (не Embedded Resource). Среда Visual Studio скомпилирует их в файл `.resources` по имени *<ИмяСборки>*`.g.resources`, в котором также содержатся скомпилированные файлы XAML (`.baml`).

Чтобы программно загрузить ресурс с ключом URI, необходимо вызвать метод `Application.GetResourceStream`:

```
Uri u = new Uri ("flag.png", UriKind.Relative);
using (Stream s = Application.GetResourceStream (u).Stream)
```

Обратите внимание на использование относительного URI. Можно также применять абсолютный URI в показанном ниже формате (три запятых подряд — это не опечатка):

```
Uri u = new Uri ("pack://application:,,,/flag.png");
```

Если вместо `Application` указать объект `Assembly`, то содержимое можно извлечь с помощью объекта `ResourceManager`:

```
Assembly a = Assembly.GetExecutingAssembly();
ResourceManager r = new ResourceManager (a.GetName().Name + ".g", a);
using (Stream s = r.GetStream ("flag.png"))
  ...
```

Объект `ResourceManager` также позволяет выполнять перечисление содержимого контейнера `.g.resources` внутри заданной сборки.

## Подчиненные сборки

Данные, встроенные в файл `.resources`, являются локализуемыми.

Проблема локализации ресурсов возникает, когда приложение запускается под управлением версии Windows, ориентированной на отображение всех элементов на другом языке. В целях согласованности приложение должно использовать тот же самый язык.

Типичная настройка предполагает наличие следующих компонентов:

- главная сборка, содержащая файлы `.resources` для стандартного или *запасного* языка;
- отдельные *подчиненные сборки*, которые содержат локализованные файлы `.resources`, переведенные на различные языки.

Когда приложение запускается, инфраструктура .NET Framework выясняет язык текущей ОС (через свойство `CultureInfo.CurrentUICulture`). Всякий раз, когда запрашивается ресурс с применением объекта `ResourceManager`, инфраструктура .NET Framework ищет локализованную подчиненную сборку. Если такая сборка доступна и содержит запрошенный ключ ресурса, тогда этот ресурс используется вместо его версии из главной сборки.

Другими словами, расширять языковую поддержку можно просто добавлением новых подчиненных сборок, не изменяя главную сборку.

Подчиненная сборка не может содержать исполняемый код — допускается наличие только ресурсов.

Подчиненные сборки развертываются в подкаталогах папки сборки, как показано ниже:

```
programBaseFolder\MyProgram.exe
                 \MyLibrary.exe
                 \XX\MyProgram.resources.dll
                 \XX\MyLibrary.resources.dll
```

Здесь *XX* — двухбуквенный код языка (скажем, `de` для немецкого) либо код языка и региона (например, `en-GB` для английского в Великобритании). Такая система именования позволяет среде CLR находить и автоматически загружать корректную подчиненную сборку.

### Построение подчиненных сборок

Вспомните предшествующий пример файла `.resx`, который имел следующее содержимое:

```
<root>
  ...
  <data name="Greeting"
    <value>hello</value>
  </data>
</root>
```

Затем во время выполнения мы извлекали приветственное сообщение (`Greeting`):

```
ResourceManager r = new ResourceManager ("welcome",
                                         Assembly.GetExecutingAssembly());
Console.Write (r.GetString ("Greeting"));
```

Предположим, что при запуске в среде немецкоязычной ОС Windows вместо "hello" требуется вывести "hallo". Первый шаг заключается в добавлении еще одного файла `.resx` по имени `welcome.de.resx`, в котором строка `hello` заменяется строкой `hallo`:

```
<root>
  <data name="Greeting">
    <value>hallo<value>
  </data>
</root>
```

В Visual Studio это все, что необходимо сделать — при компиляции в подкаталоге `de` будет автоматически создана подчиненная сборка по имени `MyApp.resources.dll`.

При работе в командной строке сначала понадобится запустить инструмент `resgen` для преобразования файла `.resx` в файл `.resources`:

```
resgen MyApp.de.resx
```

после чего запустить утилиту `al` для построения подчиненной сборки:

```
al /culture:de /out:MyApp.resources.dll /embed:MyApp.de.resources /t:lib
```

Можно указать `/template:MyApp.exe`, чтобы импортировать строгое имя главной сборки.

### Тестирование подчиненных сборок

Для эмуляции выполнения в среде ОС с другим языком потребуется изменить свойство `CurrentUICulture` с применением класса `Thread`:

```
System.Threading.Thread.CurrentThread.CurrentUICulture
  = new System.Globalization.CultureInfo ("de");
```

`CultureInfo.CurrentUICulture` — версия того же самого свойства, допускающая только чтение.

Удобная стратегия тестирования предусматривает локализацию слов, которые по-прежнему должны читаться как английские, но не использовать стандартные латинские символы Unicode (например, ℓo¢aℓizə).

### Поддержка со стороны визуальных конструкторов Visual Studio

Визуальные конструкторы в Visual Studio предлагают расширенную поддержку для локализуемых компонентов и визуальных элементов. Визуальный конструктор WPF имеет собственный рабочий поток для локализации; другие визуальные конструкторы, основанные на `Component`, применяют свойство, предназначенное только для этапа проектирования, которое показывает, что компонент или элемент управления Windows Forms имеет свойство `Language`. Для настройки на другой язык нужно просто изменить значение свойства `Language` и затем приступить к модификации компонента. Значения всех свойств элементов управления с атрибутом `Localizable` будут сохраняться в файле `.resx` для конкретного языка. Переключаться между языками можно в любой момент, просто изменяя свойство `Language`.

## Культуры и подкультуры

Культуры разделяются на собственно культуры и подкультуры. Культура представляет конкретный язык, а подкультура — региональный вариант этого языка. Инфраструктура .NET Framework следует стандарту RFC-1766, который представляет культуры и подкультуры с помощью двухбуквенных кодов. Ниже показаны коды для английской и немецкой культур:

```
en
de
```

А это коды для австралийской английской и австрийской немецкой подкультур:

```
en-AU
de-AT
```

Культура представляется в .NET с помощью класса `System.Globalization.CultureInfo`. Просмотреть текущую культуру в приложении можно следующим образом:

```
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentCulture);
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentUICulture);
```

Выполнение такого кода на компьютере, локализованном для Австралии, демонстрирует разницу между двумя указанными свойствами:

```
EN-AU
EN-US
```

Свойство `CurrentCulture` отражает региональные параметры в панели управления Windows, тогда как свойство `CurrentUICulture` указывает язык пользовательского интерфейса ОС.

Региональные параметры включают аспекты вроде часового пояса, а также форматов для валюты и дат. Свойство `CurrentCulture` определяет стандартное поведение для функций, подобных `DateTime.Parse`. Региональные параметры могут быть настроены в точке, где они больше не соответствуют какой-либо культуре.

Свойство `CurrentUICulture` определяет язык, посредством которого компьютер взаимодействует с пользователем. Австралия не нуждается в отдельной версии английского языка для данной цели, поэтому используется версия английского, принятая в США. Например, если в связи с работой приходится несколько месяцев проводить в Австрии, то имеет смысл изменить текущую культуру в панели управления на немецкий язык в Австрии. Однако в случае неумения говорить по-немецки свойство `CurrentUICulture` может остаться установленным в английский язык, принятый в США.

Для определения корректной подчиненной сборки, подлежащей загрузке, объект `ResourceManager` по умолчанию применяет свойство `CurrentUICulture` текущего потока. При загрузке ресурсов объект `ResourceManager` использует механизм обхода. Если сборка для подкультуры определена, то она и будет применяться; в противном случае будет задействована обобщенная культура. Если обобщенная культура отсутствует, тогда будет произведен возврат к стандартной культуре в главной сборке.

# Распознавание и загрузка сборок

Типичное приложение состоит из главной исполняемой сборки и набора связанных библиотечных сборок, например:

```
AdventureGame.exe
Terrain.dll
UIEngine.dll
```

*Распознавание сборок* представляет собой процесс нахождения сборок, на которые производится ссылка. Распознавание сборок происходит как на этапе компиляции, так и во время выполнения. Система, используемая на этапе компиляции, проста: компилятору известно, где искать ссылаемые сборки, потому что об этом ему было сообщено. Вы сами (или Visual Studio) указываете полный путь к ссылаемым сборкам, которые расположены не в текущем каталоге.

Распознавание во время выполнения сложнее. Компилятор записывает строгие имена ссылаемых сборок в манифест, но не предоставляет никаких подсказок о том, где их искать. В простом случае, когда все ссылаемые сборки помещаются в ту же са-

мую папку, что и главный исполняемый файл, никаких проблем не возникает, поскольку это первое место, которое будет просматривать среда CLR. Сложности возникают в следующих ситуациях:

- при развертывании ссылаемых сборок, расположенных в других местах;
- при динамической загрузке сборок.

Приложения UWP ограничены в плане возможностей настройки загрузки и распознавания сборок. В частности, загрузка сборки из произвольного местоположения не поддерживается и отсутствует событие `AssemblyResolve`.

## Правила распознавания сборок и типов

Все типы имеют область действия на уровне сборки. Сборка подобна адресу для типа. В качестве аналогии на человека можно сослаться так: "Иван" (имя типа без пространства имен), "Иван Петров" (полное имя типа) или "Иван Петров с улицы Программистов в Линуксбурге" (имя типа с указанием сборки).

На этапе компиляции для достижения уникальности вполне достаточно полного имени типа, т.к. ссылаться на две сборки, которые определяют одно и то же полное имя типа, невозможно (во всяком случае, не прибегая к специальным трюкам). Однако во время выполнения в памяти может находиться множество идентично именованных типов. Такое случается, например, внутри визуального редактора Visual Studio всякий раз, когда вы повторно компилируете спроектированные элементы. Единственный способ различения типов подобного рода — по их сборкам; следовательно, сборка формирует важную часть удостоверения типа во время выполнения. Сборка также является дескриптором типа для его кода и метаданных.

Во время выполнения среда CLR загружает сборки в момент, когда они впервые понадобились. Это происходит при ссылке на один из типов, находящихся в сборке. Например, предположим, что приложение `AdventureGame.exe` создает экземпляр типа по имени `TerrainModel.Map`. При условии, что дополнительные файлы конфигурации отсутствуют, среда CLR отвечает на перечисленные ниже вопросы.

- Каково полностью заданное имя сборки, содержащей тип `TerrainModel.Map`, когда приложение `AdventureGame.exe` было скомпилировано?
- Не загружена ли уже в память сборка с этим полностью заданным именем и таким же контекстом распознавания?

Если ответ на второй вопрос оказывается положительным, тогда среда CLR задействует существующую копию в памяти, а иначе CLR ищет сборку. Среда CLR сначала проверяет GAC, затем пути *зондирования* (обычно базовый каталог приложения) и, наконец, выдает событие `AppDomain.AssemblyResolve`. Если ни одно из действий не дало совпадения, то CLR генерирует исключение.

## Событие `AssemblyResolve`

Событие `AssemblyResolve` позволяет вмешаться в процесс и вручную загрузить сборку, которую CLR не может найти. В случае обработки события `AssemblyResolve` ссылаемые сборки можно распределять по различным местоположениям и по-прежнему обеспечивать их загрузку.

Внутри обработчика события AssemblyResolve производится поиск сборки и ее загрузка за счет вызова одного из трех статических методов класса Assembly: Load, LoadFrom или LoadFile. Указанные методы возвращают ссылку на вновь загруженную сборку, и данная ссылка затем возвращается вызывающему коду:

```
static void Main()
{
  AppDomain.CurrentDomain.AssemblyResolve += FindAssembly;
  ...
}

static Assembly FindAssembly (object sender, ResolveEventArgs args)
{
  string fullyQualifiedName = args.Name;
  Assembly a = Assembly.LoadFrom (...);
  return a;
}
```

Событие ResolveEventArgs необычно тем, что имеет возвращаемый тип. При наличии множества обработчиков преимущество получает первый обработчик, который возвратил отличный от null объект Assembly.

## Загрузка сборок

Методы Load класса Assembly удобны в применении как внутри, так и вне обработчика AssemblyResolve. За пределами этого обработчика событий методы Load могут загружать и запускать сборки, на которые не производилась ссылка во время компиляции. Примером, когда может делаться подобное, может служить запуск на выполнение подключаемого модуля.

Хорошо подумайте, прежде чем вызывать метод Load, LoadFrom или LoadFile: упомянутые методы загружают сборку в текущий домен приложения на постоянной основе – даже если никакие действия над результирующим объектом Assembly не производятся. С загрузкой сборок связан побочный эффект: она блокирует файлы сборок, а также влияет на последующее распознавание типов.

Единственный способ выгрузить сборку предусматривает выгрузку целого домена приложения. (Существует также прием, позволяющий избежать блокирования сборок, который называется *теневым копированием* для сборок в пути зондирования; читайте об этом в статье MSDN по адресу https://docs.microsoft.com/ru-ru/dotnet/framework/app-domains/shadow-copy-assemblies.)

Если нужно просто проверить сборку, не выполняя какой-либо код в ней, тогда можно использовать контекст только для рефлексии (глава 19).

Чтобы загрузить сборку с полностью заданным именем (без местоположения), вызывайте метод Assembly.Load. Он инструктирует среду CLR о необходимости поиска сборки с применением обычной автоматической системы распознавания. Среда CLR сама использует метод Load для нахождения ссылаемых сборок.

Чтобы загрузить сборку из файла, вызывайте метод LoadFrom или LoadFile.

Чтобы загрузить сборку из URI, вызывайте метод LoadFrom.

Чтобы загрузить сборку из байтового массива, вызывайте метод Load.

Для выяснения, какие сборки загружены в память в текущий момент, необходимо вызвать метод GetAssemblies класса AppDomain:

```
foreach (Assembly a in
  AppDomain.CurrentDomain.GetAssemblies())
{
  Console.WriteLine (a.Location);       // Путь к файлу
  Console.WriteLine (a.CodeBase);       // URI
  Console.WriteLine (a.GetName().Name); // Простое имя
}
```

## Загрузка из файла

Методы LoadFrom и LoadFile позволяют загружать сборку из файла с указанным именем. Они отличаются в двух отношениях. Первое отличие заключается в том, что если сборка с тем же самым удостоверением уже была загружена в память из другого местоположения, то метод LoadFrom предоставляет ее предыдущую копию:

```
Assembly a1 = Assembly.LoadFrom (@"c:\temp1\lib.dll");
Assembly a2 = Assembly.LoadFrom (@"c:\temp2\lib.dll");
Console.WriteLine (a1 == a2);  // true
```

Метод LoadFile выдает новую копию:

```
Assembly a1 = Assembly.LoadFile (@"c:\temp1\lib.dll");
Assembly a2 = Assembly.LoadFile (@"c:\temp2\lib.dll");
Console.WriteLine (a1 == a2);  // false
```

Однако в случае двукратной загрузки из *того же самого* местоположения оба метода возвращают ранее кешированную копию. (Напротив, двукратная загрузка сборки из одного и того же байтового массива предоставляет два разных объекта Assembly.)

Типы из двух идентичных сборок в памяти несовместимы. Это основная причина, по которой следует избегать загрузки дублированных сборок, а потому отдавать предпочтение методу LoadFrom перед LoadFile.

Второе отличие между методами LoadFrom и LoadFile связано с тем, что LoadFrom подсказывает среде CLR местоположение для последующих ссылок, тогда как LoadFile — нет. В целях иллюстрации предположим, что приложение в папке \folder1 загружает из папки \folder2 сборку по имени TestLib.dll, которая ссылается на сборку \folder2\Another.dll:

```
\folder1\MyApplication.exe

\folder2\TestLib.dll
\folder2\Another.dll
```

Если сборка TestLib загружается посредством метода LoadFrom, тогда среда CLR будет искать и загружать сборку Another.dll.

В случае загрузки TestLib с помощью метода LoadFile среда CLR не сможет найти сборку Another.dll и сгенерирует исключение — если только не обеспечена обработка события AssemblyResolve.

В последующих разделах мы продемонстрируем применение таких методов в контексте ряда практических приложений.

### Статически ссылаемые типы и LoadFrom/LoadFile

Когда вы ссылаетесь на тип прямо в своем коде, то на самом деле *статически ссылаетесь* на него. Компилятор встроит в компилируемую сборку ссылку на этот тип, а также имя сборки, которая его содержит (но не информацию о том, где его искать во время выполнения).

Например, пусть имеется тип по имени `Foo` в сборке `foo.dll`, а приложение `bar.exe` включает следующий код:

```
var foo = new Foo();
```

Приложение `bar.exe` статически ссылается на тип `Foo` в сборке `foo`. Взамен мы могли бы загрузить сборку `foo` динамически:

```
Type t = Assembly.LoadFrom (@"d:\temp\foo.dll").GetType ("Foo");
var foo = Activator.CreateInstance (t);
```

Когда два подхода смешиваются, в памяти обычно остаются две копии сборки, т.к. среда CLR считает, что каждая из них имеет отличающийся "контекст распознавания".

Ранее было указано, что при распознавании статических ссылок среда CLR просматривает сначала GAC, далее исследует путь зондирования (обычно базовый каталог приложения) и затем генерирует событие `AssemblyResolve`. Тем не менее, перед любым из указанных действий среда CLR проверяет, не была ли сборка уже загружена. Однако она учитывает *только* сборки, которые удовлетворяют одному из двух условий:

- сборки были загружены из пути, который иначе мог быть найден сам по себе (путь зондирования);
- сборки были загружены в ответ на поступление события `AssemblyResolve`.

Таким образом, если вы уже загрузили сборку не из пути зондирования с помощью метода `LoadFrom` или `LoadFile`, то в итоге получите в памяти две копии сборки (с несовместимыми типами). Во избежание такой ситуации при вызове `LoadFrom`/`LoadFile` следует проявлять осторожность, предварительно проверяя, существует ли сборка в базовом каталоге приложения (если только загрузка нескольких версий сборки не является *преследуемой целью*).

Загрузка в ответ на событие `AssemblyResolve` защищена от упомянутой проблемы (независимо от того, что используется – `LoadFrom`, `LoadFile` или загрузка из байтового массива, как будет показано позже), поскольку данное событие генерируется только для сборок, находящихся за пределами пути зондирования.

Какой бы метод ни применялся, `LoadFrom` или `LoadFile`, среда CLR всегда сначала ищет запрошенную сборку в GAC. Пропустить поиск в GAC можно посредством метода `ReflectionOnlyLoadFrom` (который загружает сборку в контекст, предназначенный только для рефлексии). Даже загрузка из байтового массива не пропускает поиск в GAC, хотя она обходит проблему блокировки файлов сборок:

```
byte[] image = File.ReadAllBytes (assemblyPath);
Assembly a = Assembly.Load (image);
```

Если вы поступаете так, то должны обрабатывать событие `AssemblyResolve` класса `AppDomain`, чтобы распознать любые сборки, на которые ссылается сама загруженная сборка, и отслеживать все загруженные сборки (как описано в разделе "Упаковка однофайловой исполняемой сборки" далее в главе).

### Сравнение свойств Location и CodeBase

Свойство Location класса Assembly обычно возвращает физическое местоположение сборки в файловой системе (если она там присутствует). Свойство CodeBase отражает местоположение в форме URI, исключая специальные случаи, такие как ситуация, когда сборка загружена из Интернета, при которой CodeBase хранит URI в Интернете, а Location – временный путь внутри файловой системы, куда сборка была загружена. Другой специальный случай касается сборок *теневого копирования*, при котором свойство Location является пустым, а свойство CodeBase указывает на местоположение, не связанное с теневым копированием. Технология ASP.NET и популярная инфраструктура тестирования NUnit задействуют теневое копирование, чтобы позволить сборкам обновляться во время функционирования веб-сайта или выполнения модульных тестов (соответствующая статья MSDN доступна по адресу https://docs.microsoft.com/ru-ru/dotnet/framework/app-domains/shadow-copy-assemblies). Инструмент LINQPad делает нечто подобное при ссылке на специальные сборки.

Следовательно, если вы ищете местоположение сборки на диске, то полагаться на одно лишь свойство Location опасно. Более надежный подход предполагает проверку обоих свойств. Приведенный далее метод возвращает папку, содержащую сборку (или null, если выяснить ее невозможно):

```
public static string GetAssemblyFolder (Assembly a)
{
  try
  {
    if (!string.IsNullOrEmpty (a.Location))
      return Path.GetDirectoryName (a.Location);

    if (string.IsNullOrEmpty (a.CodeBase)) return null;

    var uri = new Uri (a.CodeBase);
    if (!uri.IsFile) return null;

    return Path.GetDirectoryName (uri.LocalPath);
  }
  catch (NotSupportedException)
  {
    return null;      // Динамическая сборка, сгенерированная с помощью
                      // пространства имен Reflection.Emit
  }
}
```

Обратите внимание, что поскольку свойство CodeBase возвращает URI, для получения пути к локальному файлу мы используем класс Uri.

# Развертывание сборок за пределами базовой папки

Иногда сборки необходимо развертывать в местоположениях, отличающихся от базового каталога приложения, например:

```
..\MyProgram\Main.exe
..\MyProgram\Libs\V1.23\GameLogic.dll
..\MyProgram\Libs\V1.23\3DEngine.dll
..\MyProgram\Terrain\Map.dll
..\Common\TimingController.dll
```

В таком случае среде CLR потребуется помочь находить сборки, располагающиеся за пределами базовой папки. Простейшее решение заключается в обработке события AssemblyResolve. В следующем примере мы предполагаем, что все дополнительные сборки содержатся в каталоге c:\ExtraAssemblies:

```
using System;
using System.IO;
using System.Reflection;

class Loader
{
  static void Main()
  {
    AppDomain.CurrentDomain.AssemblyResolve += FindAssembly;

    // Перед попыткой использования любых типов в c:\ExtraAssemblies
    // мы должны переключиться на другой класс:
    Program.Go();
  }

  static Assembly FindAssembly (object sender, ResolveEventArgs args)
  {
    string simpleName = new AssemblyName (args.Name).Name;
    string path = @"c:\ExtraAssemblies\" + simpleName + ".dll";

    if (!File.Exists (path)) return null;     // Контроль корректности
    return Assembly.LoadFrom (path);          // Загрузка
  }
}

class Program
{
  internal static void Go()
  {
    // Теперь мы можем ссылаться на типы, определенные в c:\ExtraAssemblies
  }
}
```

В приведенном примере жизненно важно не ссылаться на типы в каталоге c:\ExtraAssemblies напрямую из класса Loader (скажем, как на поля), потому что тогда среда CLR попытается распознать эти типы перед попаданием в метод Main.

В приведенном примере мы могли бы применять либо метод LoadFrom, либо метод LoadFile. В любом случае среда CLR проверяет, имеет ли обрабатываемая сборка в точности запрошенное удостоверение, что обеспечивает целостность строго именованных ссылок.

В главе 24 мы опишем другой подход, который можно использовать, когда создаются новые домены приложений. Он предусматривает установку свойства PrivateBinPath домена приложения для включения каталогов, содержащих дополнительные сборки, что расширяет стандартные местоположения зондирования сборок. Ограничение такого подхода связано с тем, что все дополнительные каталоги должны находиться *ниже* базового каталога приложения.

# Упаковка однофайловой исполняемой сборки

Предположим, что вы построили приложение, состоящее из десяти сборок: один главный исполняемый файл и девять DLL-библиотек. Хотя такой уровень детализации может быть великолепным для проектирования и отладки, неплохо также иметь возможность упаковки всех сборок в единственный исполняемый файл вида "щелкнуть и запустить", не требуя от пользователя выполнения какой-то процедуры установки либо извлечения файлов. Для этого скомпилированные DLL-библиотеки сборок можно включить в проект главного исполняемого файла как встроенные ресурсы и затем написать обработчик событий `AssemblyResolve`, обеспечивающий загрузку двоичных образов DLL-библиотек по требованию. Ниже показано, как следует поступить.

```
using System;
using System.IO;
using System.Reflection;
using System.Collections.Generic;

public class Loader
{
  static Dictionary <string, Assembly> _libs
   = new Dictionary <string, Assembly>();

  static void Main()
  {
    AppDomain.CurrentDomain.AssemblyResolve += FindAssembly;
    Program.Go();
  }

  static Assembly FindAssembly (object sender, ResolveEventArgs args)
  {
    string shortName = new AssemblyName (args.Name).Name;
    if (_libs.ContainsKey (shortName)) return _libs [shortName];

    using (Stream s = Assembly.GetExecutingAssembly().
          GetManifestResourceStream ("Libs." + shortName + ".dll"))
    {
      byte[] data = new BinaryReader (s).ReadBytes ((int) s.Length);
      Assembly a = Assembly.Load (data);
      _libs [shortName] = a;
      return a;
    }
  }
}

public class Program
{
  public static void Go()
  {
    // Запустить главную программу...
  }
}
```

Так как класс `Loader` определен в главной исполняемой сборке, вызов метода `Assembly.GetExecutingAssembly` будет всегда возвращать сборку главного исполняемого файла, куда были включены скомпилированные DLL-библиотеки в виде встроенных ресурсов. В рассматриваемом примере имя каждой сборки, являющейся встроенным ресурсом, предваряется префиксом `"Libs."`. Если применялась IDE-среда Visual

Studio, то "Libs." можно было бы заменить стандартным пространством имен проекта (на вкладке Application (Приложение) окна свойств проекта). Может также возникнуть необходимость удостовериться в том, что свойство Build Action (Действие построения) для каждой DLL-библиотеки установлено в Embedded Resource (Встроенный ресурс).

Причина кеширования запрошенных сборок в словаре связана с обеспечением идентичности возвращаемых объектов в ситуации, когда среда CLR запрашивает ту же самую сборку снова. В противном случае типы сборки окажутся несовместимыми с типами этой же сборки, которая была загружена ранее (несмотря на идентичность их двоичных образов).

В качестве вариации ссылаемые сборки можно сжимать на этапе компиляции, а затем распаковывать в методе FindAssembly с использованием класса DeflateStream.

# Работа со сборками, не имеющими ссылок на этапе компиляции

Иногда удобно явно загружать сборки .NET, на которые могут отсутствовать ссылки на этапе компиляции.

Если данная сборка является исполняемым файлом, который нужно просто запустить, тогда достаточно вызвать метод ExecuteAssembly на текущем домене приложения. Метод ExecuteAssembly загружает исполняемый файл с применением семантики метода LoadFrom и затем вызывает его метод точки входа с необязательными аргументами командной строки. Например:

```
string dir = AppDomain.CurrentDomain.BaseDirectory;
AppDomain.CurrentDomain.ExecuteAssembly (Path.Combine (dir, "test.exe"));
```

Метод ExecuteAssembly работает синхронным образом, т.е. вызывающий метод блокируется до тех пор, пока выполнение вызванной сборки не завершится. Для асинхронной работы метод ExecuteAssembly должен быть вызван в другом потоке или задаче (см. главу 14).

Тем не менее, в большинстве случаев сборка, которую необходимо загрузить, будет библиотекой. Используемый подход заключается в вызове метода LoadFrom и работе с типами сборки посредством рефлексии. Ниже приведен пример:

```
string ourDir = AppDomain.CurrentDomain.BaseDirectory;
string plugInDir = Path.Combine (ourDir, "plugins");
Assembly a = Assembly.LoadFrom (Path.Combine (plugInDir, "widget.dll"));
Type t = a.GetType ("Namespace.TypeName");
object widget = Activator.CreateInstance (t);  // (См. главу 19.)
...
```

Здесь применяется метод LoadFrom, а не LoadFile, чтобы обеспечить загрузку из той же папки любых закрытых сборок, на которые ссылается widget.dll. Затем из сборки извлекается нужный тип по имени и создается его экземпляр.

На следующем шаге можно было бы воспользоваться рефлексией для динамического вызова методов и доступа к свойствам объекта widget; мы покажем, как это делать, в следующей главе. Более простой — и быстрый — подход предусматривает приведение такого объекта к типу, который воспринимается обеими сборками. Часто им будет интерфейс, определенный в какой-то общей сборке:

```
public interface IPluggable
{
  void ShowAboutBox();
  ...
}
```

Теперь имеется возможность поступать так:

```
Type t = a.GetType ("Namespace.TypeName");
IPluggable widget = (IPluggable) Activator.CreateInstance (t);
widget.ShowAboutBox();
```

Похожую систему можно применять для динамически публикуемых служб в инфраструктуре WCF или на сервере Remoting. В представленном далее коде предполагается, что имена библиотек, к которым необходимо открыть доступ, заканчиваются на `Server`:

```
using System.IO;
using System.Reflection;
...
string dir = AppDomain.CurrentDomain.BaseDirectory;
foreach (string assFile in Directory.GetFiles (dir, "*Server.dll"))
{
  Assembly a = Assembly.LoadFrom (assFile);
  foreach (Type t in a.GetTypes())
    if (typeof (MyBaseServerType).IsAssignableFrom (t))
    {
      // Открыть доступ к типу t
    }
}
```

Однако такой подход приводит к тому, что станет очень легко добавлять мошеннические или дефектные сборки, возможно, даже непредумышленно! Предполагая, что ссылки на этапе компиляции отсутствуют, среда CLR не предпринимает никаких действий по проверке удостоверений таких сборок. Если загружаемые сборки подписаны известным открытым ключом, то решением будет явная проверка ключа. В следующем примере мы предполагаем, что все библиотеки подписаны с помощью той же самой пары ключей, что и выполняемая сборка:

```
byte[] ourPK = Assembly.GetExecutingAssembly().GetName().GetPublicKey();
foreach (string assFile in Directory.GetFiles (dir, "*Server.dll"))
{
  byte[] targetPK = AssemblyName.GetAssemblyName (assFile).GetPublicKey();
  if (Enumerable.SequenceEqual (ourPK, targetPK))
  {
    Assembly a = Assembly.LoadFrom (assFile);
    ...
```

Обратите внимание на то, что класс `AssemblyName` позволяет проверить открытый ключ *перед* загрузкой сборки. Для сравнения байтовых массивов используется метод `SequenceEqual` из LINQ (`System.Linq`).

# 19

# Рефлексия и метаданные

Как было показано в предыдущей главе, программа на языке C# компилируется в сборку, которая содержит метаданные, скомпилированный код и ресурсы. Процесс инспектирования метаданных и скомпилированного кода во время выполнения называется *рефлексией*.

Скомпилированный код в сборке включает почти все содержимое первоначального исходного кода. Некоторая информация утрачивается, например, имена локальных переменных, комментарии и директивы препроцессора. Тем не менее, рефлексия позволяет получить доступ практически ко всему остальному, даже делая возможным написание декомпилятора.

Многие службы, доступные в .NET и открытые через язык C# (такие как динамическое связывание, сериализация, привязка данных и удаленная обработка (Remoting)), полагаются на присутствие метаданных. Ваши программы также могут использовать метаданные в своих интересах и даже расширять их новой информацией, применяя специальные атрибуты. В пространстве имен `System.Reflection` находится API-интерфейс рефлексии. Кроме того, с помощью классов из пространства имен `System.Reflection.Emit` во время выполнения можно динамически создавать новые метаданные и исполняемые инструкции на промежуточном языке (Intermediate Language – IL).

В примерах настоящей главы предполагается, что вы импортировали пространства имен `System` и `System.Reflection`, а также `System.Reflection.Emit`.

Когда мы используем в главе термин "динамическое", то имеем в виду применение рефлексии для исполнения задачи, для которой безопасность типов обеспечивается только во время выполнения. По принципу это похоже на *динамическое связывание* посредством ключевого слова `dynamic` в C#, хотя механизм и функциональность здесь другие.

Сравнивая две технологии, можно сказать, что динамическое связывание намного проще в использовании и применяет среду DLR для взаимодействия с динамическими языками. Рефлексия относительно неудобна в использовании, связана только со средой CLR, но обладает большей гибкостью в плане того, что можно делать с помощью CLR. Например, рефлексия позволяет получать списки типов и членов, создавать объекты, имена которых указываются в виде строк, и строить сборки на лету.

# Рефлексия и активизация типов

В этом разделе мы рассмотрим, как получать экземпляр класса Type, инспектировать его метаданные и использовать его для динамического создания объекта.

## Получение экземпляра Type

Экземпляр класса System.Type представляет метаданные для типа. Поскольку класс Type применяется очень широко, он находится в пространстве имен System, а не в System.Reflection.

Получить экземпляр System.Type можно путем вызова метода GetType на любом объекте или с помощью операции typeof языка C#:

```
Type t1 = DateTime.Now.GetType(); //Экземпляр Type, полученный во время выполнения
Type t2 = typeof (DateTime);  // Экземпляр Type, полученный на этапе компиляции
```

Операцию typeof можно использовать для получения типов массивов и обобщенных типов:

```
Type t3 = typeof (DateTime[]);              // Тип одномерного массива
Type t4 = typeof (DateTime[,]);             // Тип двухмерного массива
Type t5 = typeof (Dictionary<int,int>);     // Закрытый обобщенный тип
Type t6 = typeof (Dictionary<,>);           // Несвязанный обобщенный тип
```

Экземпляр Type можно также извлекать по имени. При наличии ссылки на его сборку (Assembly) необходимо вызвать метод Assembly.GetType (как будет описано более подробно в разделе "Рефлексия сборок" далее в главе):

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

Если объект Assembly отсутствует, то тип можно получить через его *имя с указанием сборки* (полное имя типа, за которым следует полностью заданное имя сборки). Сборка неявно загружается, как если бы вызывался метод Assembly.Load(string):

```
Type t = Type.GetType ("System.Int32, mscorlib, Version=2.0.0.0, " +
                       "Culture=neutral, PublicKeyToken=b77a5c561934e089");
```

Имея объект System.Type, его свойства можно применять для доступа к имени типа, сборке, базовому типу, видимости и т.д. Например:

```
Type stringType = typeof (string);
string name     = stringType.Name;          // String
Type baseType   = stringType.BaseType;      // typeof(Object)
Assembly assem  = stringType.Assembly;      // mscorlib.dll
bool isPublic   = stringType.IsPublic;      // true
```

Экземпляр System.Type — своего рода окно в мир метаданных для этого типа, а также для сборки, в которой он определен.

Класс System.Type является абстрактным, так что операция typeof должна в действительности давать подкласс класса Type. Среда CLR использует внутренний подкласс сборки mscorlib по имени RuntimeType.

### TypeInfo и приложения Windows Store

Если вы планируете ориентироваться на более старый профиль Windows Store, то обнаружите, что большинство членов Type отсутствует. Взамен доступ к отсутствующим членам открывается через класс по имени TypeInfo, экземпляр которого получается вызовом GetTypeInfo. Таким образом, чтобы заставить код предыдущего примера выполняться, вот как нужно поступить:

```
Type stringType = typeof(string);
string name = stringType.Name;
Type baseType = stringType.GetTypeInfo().BaseType;
Assembly assem = stringType.GetTypeInfo().Assembly;
bool isPublic = stringType.GetTypeInfo().IsPublic;
```

Код во многих листингах в настоящей главе требует такой модификации, чтобы работать в более старом профиле Windows Store. Следовательно, если пример не компилируется из-за отсутствия члена, тогда необходимо добавить к выражению Type конструкцию .GetTypeInfo().

Класс TypeInfo также существует в полной версии .NET Framework, поэтому код, работающий в приложениях Windows Store, будет функционировать и в настольных приложениях, ориентированных на .NET Framework 4.5 и последующие версии. Класс TypeInfo также включает дополнительные свойства и методы для выполнения рефлексии членов.

Приложения UWP и .NET Core ограничены в том, что они могут делать в плане рефлексии. В них запрещены некоторые действия, такие как доступ к неоткрытым членам или создание типов с помощью Reflection.Emit.

### Получение типов массивов

Как только что было указано, операция typeof и метод GetType имеют дело с типами массивов. Получить тип массива можно также за счет вызова метода MakeArrayType на типе *элементов* массива:

```
Type simpleArrayType = typeof (int).MakeArrayType();
Console.WriteLine (simpleArrayType == typeof (int[]));    // True
```

Методу MakeArrayType можно передать целочисленный аргумент, чтобы создать многомерный прямоугольный массив:

```
Type cubeType = typeof (int).MakeArrayType (3);           // В форме куба
Console.WriteLine (cubeType == typeof (int[,,]));         // True
```

Метод GetElementType делает обратное — извлекает тип элементов массива:

```
Type e = typeof (int[]).GetElementType();  // e == typeof (int)
```

Метод GetArrayRank возвращает количество измерений в многомерном массиве:

```
int rank = typeof (int[,,]).GetArrayRank();               // 3
```

### Получение вложенных типов

Чтобы извлечь вложенные типы, нужно вызвать метод GetNestedTypes на содержащем их типе. Например:

```
foreach (Type t in typeof (System.Environment).GetNestedTypes())
  Console.WriteLine (t.FullName);

ВЫВОД: System.Environment+SpecialFolder
```

Или в приложении Windows Store:

```
foreach (TypeInfo t in typeof (System.Environment).GetTypeInfo()
                                                  .DeclaredNestedTypes)
  Debug.WriteLine (t.FullName);
```

С вложенными типами связано одно предостережение: среда CLR трактует вложенный тип как имеющий специальные "вложенные" уровни доступности, например:

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine (t.IsPublic);            // False
Console.WriteLine (t.IsNestedPublic);      // True
```

## Имена типов

Тип имеет свойства `Namespace`, `Name` и `FullName`. В большинстве случаев `FullName` является объединением первых двух свойств:

```
Type t = typeof (System.Text.StringBuilder);

Console.WriteLine (t.Namespace);  // System.Text
Console.WriteLine (t.Name);       // StringBuilder
Console.WriteLine (t.FullName);   // System.Text.StringBuilder
```

Из указанного правила существуют два исключения: вложенные типы и закрытые обобщенные типы.

Класс `Type` также имеет свойство по имени `AssemblyQualifiedName`, возвращающее значение свойства `FullName`, за которым следует запятая и полное имя сборки. Это та самая строка, которую можно передавать методу `Type.GetType`, и она уникальным образом идентифицирует тип внутри стандартного контекста загрузки.

### Имена вложенных типов

В случае вложенных типов содержащий тип присутствует только в `FullName`:

```
Type t = typeof (System.Environment.SpecialFolder);

Console.WriteLine (t.Namespace);  // System
Console.WriteLine (t.Name);       // SpecialFolder
Console.WriteLine (t.FullName);   // System.Environment+SpecialFolder
```

Символ + отделяет содержащий тип от вложенного пространства имен.

### Имена обобщенных типов

Имена обобщенных типов снабжаются суффиксами в виде символа ', за которым следует количество параметров типа. Если обобщенный тип является несвязанным, то такое правило применяется и к `Name`, и к `FullName`:

```
Type t = typeof (Dictionary<,>);   // Unbound (несвязанный)
Console.WriteLine (t.Name);        // Dictionary'2
Console.WriteLine (t.FullName);    // System.Collections.Generic.Dictionary'2
```

Однако если обобщенный тип является закрытым, то свойство `FullName` (единственное) приобретает важное дополнение: список всех параметров типа, для каждого из которых указывается полное *имя, включающее сборку*:

```
Console.WriteLine (typeof (Dictionary<int,string>).FullName);
```

```
// ВЫВОД:
System.Collections.Generic.Dictionary'2[[System.Int32, mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089],
[System.String, mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]]
```

В итоге гарантируется, что свойство `AssemblyQualifiedName` (комбинация полного имени типа и имени сборки) содержит достаточный объем информации для исчерпывающей идентификации как обобщенного типа, так и его параметров типа.

### Имена типов массивов и указателей

Массивы представляются с тем же суффиксом, который используется в выражении `typeof`:

```
Console.WriteLine (typeof ( int[]  ).Name);      // Int32[]
Console.WriteLine (typeof ( int[,] ).Name);      // Int32[,]
Console.WriteLine (typeof ( int[,] ).FullName); // System.Int32[,]
```

Типы указателей подобны:

```
Console.WriteLine (typeof (byte*).Name);         // Byte*
```

### Имена типов параметров `ref` и `out`

Экземпляр `Type`, описывающий параметр `ref` или `out`, имеет суффикс `&`:

```
Type t = typeof (bool).GetMethod ("TryParse").GetParameters()[1]
                                             .ParameterType;
Console.WriteLine (t.Name);  // Boolean&
```

Более подробно об этом речь пойдет в разделе "Рефлексия и вызов членов" далее в главе.

## Базовые типы и интерфейсы

Класс `Type` открывает доступ к свойству `BaseType`:

```
Type base1 = typeof (System.String).BaseType;
Type base2 = typeof (System.IO.FileStream).BaseType;

Console.WriteLine (base1.Name);    // Object
Console.WriteLine (base2.Name);    // Stream
```

Метод `GetInterfaces` возвращает интерфейсы, которые тип реализует:

```
foreach (Type iType in typeof (Guid).GetInterfaces())
  Console.WriteLine (iType.Name);

IFormattable
IComparable
IComparable'1
IEquatable'1
```

Рефлексия предоставляет два динамических эквивалента статической операции `is` языка C#.

- `IsInstanceOfType`. Принимает тип и экземпляр.
- `IsAssignableFrom`. Принимает два типа.

Вот пример применения первого метода:

```
object obj    = Guid.NewGuid();
Type target   = typeof (IFormattable);

bool isTrue   = obj is IFormattable;            // Статическая операция C#
bool alsoTrue = target.IsInstanceOfType (obj);  // Динамический эквивалент
```

Метод `IsAssignableFrom` более универсален:

```
Type target = typeof (IComparable), source = typeof (string);
Console.WriteLine (target.IsAssignableFrom (source));  // True
```

Метод `IsSubclassOf` работает по тому же самому принципу, что и `IsAssignableFrom`, но исключает интерфейсы.

## Создание экземпляров типов

Динамически создать объект из его типа можно двумя способами:

- вызвать статический метод `Activator.CreateInstance`;
- вызвать метод `Invoke` на объекте `ConstructorInfo`, который получен в результате вызова метода `GetConstructor` на экземпляре `Type` (расширенные сценарии).

Метод `Activator.CreateInstance` принимает экземпляр `Type` и дополнительные аргументы, передаваемые конструктору:

```
int i = (int) Activator.CreateInstance (typeof (int));

DateTime dt = (DateTime) Activator.CreateInstance (typeof (DateTime),
                                                   2000, 1, 1);
```

Метод `CreateInstance` позволяет указывать многие другие данные, такие как сборка, из которой загружается тип, целевой домен приложения и необходимость привязки к неоткрытому конструктору. Если исполняющей среде не удается найти подходящий конструктор, то генерируется исключение `MissingMethodException`.

Вызов метода `Invoke` класса `ConstructorInfo` нужен, когда значения аргументов не позволяют устранить неоднозначность между перегруженными конструкторами. Например, пусть класс `X` имеет два конструктора: один принимает параметр типа `string`, а другой — параметр типа `StringBuilder`. В случае передачи аргумента `null` методу `Activator.CreateInstance` выбор целевого конструктора будет неоднозначным. В такой ситуации взамен должен использоваться класс `ConstructorInfo`:

```
// Извлечь конструктор, который принимает единственный параметр типа string:
ConstructorInfo ci = typeof (X).GetConstructor (new[] { typeof (string) });

// Сконструировать объект с применением перегруженной версии,
// передавая значение null:
object foo = ci.Invoke (new object[] { null });
```

Или при нацеливании на более старый профиль Windows Store:

```
ConstructorInfo ci = typeof (X).GetTypeInfo().DeclaredConstructors
  .FirstOrDefault (c =>
    c.GetParameters().Length == 1 &&
    c.GetParameters()[0].ParameterType == typeof (string));
```

Чтобы получить неоткрытый конструктор, потребуется указать соответствующее значение перечисления `BindingFlags` — данный вопрос обсуждается в подразделе "Доступ к неоткрытым членам" раздела "Рефлексия и вызов членов" далее в главе.

Динамическое создание экземпляров добавляет несколько микросекунд ко времени, которое занимает конструирование объекта. В относительном выражении это довольно много, потому что CLR обычно создает объекты очень быстро (выполнение простой операции new на небольшом классе требует нескольких десятков наносекунд).

Чтобы динамически создать объект массива на основе только типа его элементов, сначала понадобится вызвать метод MakeArrayType. Можно также создавать экземпляры обобщенных типов, как будет показано в следующем разделе.

Для динамического создания объекта делегата необходимо вызвать метод Delegate.CreateDelegate. Ниже приведен пример, демонстрирующий создание делегата экземпляра и статического делегата:

```
class Program
{
  delegate int IntFunc (int x);

  static int Square (int x) { return x * x; }          // Статический метод
  int        Cube   (int x) { return x * x * x; }      // Метод экземпляра

  static void Main()
  {
    Delegate staticD = Delegate.CreateDelegate
      (typeof (IntFunc), typeof (Program), "Square");

    Delegate instanceD = Delegate.CreateDelegate
      (typeof (IntFunc), new Program(), "Cube");

    Console.WriteLine (staticD.DynamicInvoke (3));    // 9
    Console.WriteLine (instanceD.DynamicInvoke (3))   // 27
  }
}
```

Запустить возвращенный объект Delegate можно за счет вызова метода DynamicInvoke, как делалось в показанном примере, либо путем приведения к типизированному делегату:

```
IntFunc f = (IntFunc) staticD;
Console.WriteLine (f(3));  // 9 (но выполняется намного быстрее!)
```

Вместо имени метода в CreateDelegate можно передать объект MethodInfo. Мы опишем класс MethodInfo в разделе "Рефлексия и вызов членов" далее в главе вместе с обоснованием приведения динамически созданного делегата обратно к типу статического делегата.

## Обобщенные типы

Класс Type способен представлять закрытый или несвязанный обобщенный тип. Как и на этапе компиляции, экземпляр закрытого обобщенного типа может быть создан, а экземпляр несвязанного обобщенного типа — нет:

```
Type closed = typeof (List<int>);
List<int> list = (List<int>) Activator.CreateInstance (closed);  // Допускается

Type unbound   = typeof (List<>);
object anError = Activator.CreateInstance (unbound); //Ошибка времени выполнения
```

Метод MakeGenericType преобразует несвязанный обобщенный тип в закрытый. Необходимо просто передать желаемые аргументы типа:

```
Type unbound = typeof (List<>);
Type closed = unbound.MakeGenericType (typeof (int));
```

Метод `GetGenericTypeDefinition` делает противоположное:

```
Type unbound2 = closed.GetGenericTypeDefinition();  // unbound == unbound2
```

Свойство `IsGenericType` возвращает `true`, если экземпляр `Type` является обобщенным, а свойство `IsGenericTypeDefinition` возвращает `true`, если обобщенный тип несвязанный. Приведенный далее код проверяет, является ли указанный тип типом значения, допускающим `null`:

```
Type nullable = typeof (bool?);
Console.WriteLine (
  nullable.IsGenericType &&
  nullable.GetGenericTypeDefinition() == typeof (Nullable<>));  // True
```

Метод `GetGenericArguments` возвращает аргументы типа для закрытых обобщенных типов:

```
Console.WriteLine (closed.GetGenericArguments()[0]);     // System.Int32
Console.WriteLine (nullable.GetGenericArguments()[0]);   // System.Boolean
```

Для несвязанных обобщенных типов метод `GetGenericArguments` возвращает псевдотипы, которые представляют типы-заполнители, указанные в определениях обобщенных типов:

```
Console.WriteLine (unbound.GetGenericArguments()[0]);  // T
```

Во время выполнения все обобщенные типы будут либо *несвязанными*, либо *закрытыми*. Они оказываются несвязанными в (относительно редком) случае такого выражения, как `typeof(Foo<>)`; иначе они закрыты. Во время выполнения нет понятия *открытого* обобщенного типа: все открытые типы закрываются компилятором. Метод `Test` в следующем классе всегда выводит `False`:

```
class Foo<T>
{
  public void Test()
  {
    Console.Write (GetType().IsGenericTypeDefinition);
  }
}
```

# Рефлексия и вызов членов

Метод `GetMembers` возвращает члены типа. Взгляните на показанный ниже класс:

```
class Walnut
{
  private bool cracked;
  public void Crack() { cracked = true; }
}
```

Выполнить рефлексию его открытых членов можно следующим образом:

```
MemberInfo[] members = typeof (Walnut).GetMembers();
foreach (MemberInfo m in members)
  Console.WriteLine (m);
```

Вот результат:

```
Void Crack()
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
Void .ctor()
```

### Выполнение рефлексии членов с помощью TypeInfo

Класс TypeInfo открывает доступ к другому (и в чем-то более простому) протоколу для проведения рефлексии членов. Использовать данный API-интерфейс в приложениях, ориентированных на .NET Framework 4.5 и последующие версии, не обязательно, однако обязательно для более старых приложений Windows Store, т.к. точный эквивалент метода GetMembers в них отсутствует.

Вместо открытия доступа к методам, подобным GetMembers, который возвращает массивы, класс TypeInfo предлагает *свойства*, возвращающие объекты IEnumerable<T>, на которых обычно запускаются запросы LINQ. Самым широко применяемым свойством является DeclaredMembers:

```
IEnumerable<MemberInfo> members =
  typeof(Walnut).GetTypeInfo().DeclaredMembers;
```

В отличие от метода GetMembers из результата исключены унаследованные члены:

```
Void Crack()
Void .ctor()
Boolean cracked
```

Предусмотрены также свойства для возвращения специфических разновидностей членов (DeclaredProperties, DeclaredMethods, DeclaredEvents и т.д.) и методы для возвращения конкретных членов по именам (например, GetDeclaredMethod). Последние не могут применяться для перегруженных методов (поскольку нет никакого способа указать типы параметров). Взамен в отношении свойства DeclaredMethods запускается запрос LINQ:

```
MethodInfo method = typeof (int).GetTypeInfo().DeclaredMethods
  .FirstOrDefault (m => m.Name == "ToString" &&
                        m.GetParameters().Length == 0);
```

При вызове без аргументов метод GetMembers возвращает все открытые члены для типа (и его базовых типов). Метод GetMember извлекает отдельный член по имени, хотя и возвращает массив, т.к. члены могут быть перегруженными:

```
MemberInfo[] m = typeof (Walnut).GetMember ("Crack");
Console.WriteLine (m[0]);  // Void Crack()
```

В классе MemberInfo также имеется свойство по имени MemberType типа MemberTypes, который представляет собой перечисление флагов со следующими значениями:

```
All             Custom        Field         NestedType        TypeInfo
Constructor     Event         Method        Property
```

При вызове методу GetMembers можно передать экземпляр MemberTypes, чтобы ограничить виды возвращаемых членов. В качестве альтернативы допускается ограничивать результирующий набор, вызывая методы GetMethods, GetFields,

GetProperties, GetEvents, GetConstructors и GetNestedTypes. Для каждого из перечисленных методов доступны также версии с именами в единственном числе, позволяющие получать конкретный член.

При извлечении члена типа полезно придерживаться максимально возможной конкретизации, чтобы работа кода не нарушалась, если позже будут добавлены дополнительные члены. Если осуществляется извлечение метода по имени, тогда указание типов всех параметров гарантирует, что код сохранит работоспособность и после перегрузки метода в будущем (примеры будут приведены в разделе "Параметры методов" далее в главе).

Объект MemberInfo имеет свойство Name и два свойства, возвращающие экземпляр Type.

- DeclaringType. Возвращает экземпляр Type, который определяет член.
- ReflectedType. Возвращает экземпляр Type, на котором был вызван метод GetMembers.

Эти два свойства отличаются при вызове на члене, который определен в базовом типе: DeclaringType возвращает базовый тип, тогда как ReflectedType — подтип, что отражено в следующем примере:

```
class Program
{
  static void Main()
  {
    // MethodInfo - это подкласс MemberInfo; см. рис. 19.1.

    MethodInfo test = typeof (Program).GetMethod ("ToString");
    MethodInfo obj  = typeof (object) .GetMethod ("ToString");

    Console.WriteLine (test.DeclaringType);      // System.Object
    Console.WriteLine (obj.DeclaringType);       // System.Object

    Console.WriteLine (test.ReflectedType);      // Program
    Console.WriteLine (obj.ReflectedType);       // System.Object

    Console.WriteLine (test == obj);             // False
  }
}
```

Так как объекты test и obj имеют разные значения в свойстве ReflectedType, они не равны. Тем не менее, отличие между ними — чистая "выдумка" API-интерфейса рефлексии; тип Program не имеет отдельного метода ToString во внутренней системе типов. Мы можем удостовериться в том, что эти два объекта MethodInfo ссылаются на тот же самый метод, одним из двух способов:

```
Console.WriteLine (test.MethodHandle == obj.MethodHandle);    // True

Console.WriteLine (test.MetadataToken == obj.MetadataToken
                && test.Module == obj.Module);                // True
```

Свойство MethodHandle уникально для каждого (по-настоящему отличающегося) метода внутри домена приложения, а свойство MetadataToken уникально среди всех типов и членов в рамках модуля сборки.

В классе MemberInfo также определены методы для возвращения специальных атрибутов (они рассматриваются в разделе "Извлечение атрибутов во время выполнения" далее в главе).

Получить объект `MethodBase` текущего выполняющегося метода можно путем вызова статического метода `MethodBase.GetCurrentMethod`.

## Типы членов

Сам класс `MemberInfo` в плане членов легковесен, потому что он является абстрактным базовым классом для типов, показанных на рис. 19.1.

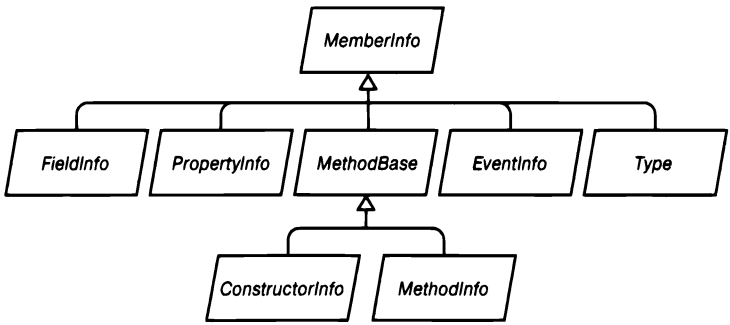



*Рис. 19.1. Типы членов*

Экземпляр класса `MemberInfo` можно приводить к его подтипам на основе свойства `MemberType`. Если член получен через методы `GetMethod`, `GetField`, `GetProperty`, `GetEvent`, `GetConstructor` или `GetNestedType` (либо с помощью их версий с именами во множественном числе), тогда приведение не обязательно. В табл. 19.1 показано, какие методы должны использоваться для всех видов конструкций языка C#.

**Таблица 19.1. Извлечение метаданных членов**

| Конструкция C# | Используемый метод | Используемое имя | Результат |
|---|---|---|---|
| Метод | `GetMethod` | (Имя метода) | `MethodInfo` |
| Свойство | `GetProperty` | (Имя свойства) | `PropertyInfo` |
| Индексатор | `GetDefaultMembers` | | `MemberInfo[]` (массив, содержащий объекты `PropertyInfo`, если скомпилирован в C#) |
| Поле | `GetField` | (Имя поля) | `FieldInfo` |
| Член перечисления | `GetField` | (Имя члена) | `FieldInfo` |
| Событие | `GetEvent` | (Имя события) | `EventInfo` |
| Конструктор | `GetConstructor` | | `ConstructorInfo` |
| Финализатор | `GetMethod` | `"Finalize"` | `MethodInfo` |
| Операция | `GetMethod` | `"op_"` + имя операции | `MethodInfo` |
| Вложенный тип | `GetNestedType` | (Имя типа) | `Type` |

Каждый подкласс MemberInfo имеет множество свойств и методов, которые отражают все аспекты метаданных члена, в том числе видимость, модификаторы, аргументы обобщенных типов, параметры, возвращаемый тип и специальные атрибуты.

Ниже демонстрируется применение метода GetMethod:

```
MethodInfo m = typeof (Walnut).GetMethod ("Crack");
Console.WriteLine (m);                  // Void Crack()
Console.WriteLine (m.ReturnType);       // System.Void
```

Все экземпляры *Info кешируются API-интерфейсом рефлексии при первом использовании:

```
MethodInfo method = typeof (Walnut).GetMethod ("Crack");
MemberInfo member = typeof (Walnut).GetMember ("Crack") [0];
Console.Write (method == member);       // True
```

Кроме предохранения идентичности объектов кеширование улучшает показатели производительности в противном случае довольно медленно работающего API-интерфейса рефлексии.

## Сравнение членов C# и членов CLR

В табл. 19.1 видно, что некоторые функциональные конструкции C# не имеют однозначного соответствия с конструкциями CLR. Причина в том, что среда CLR и API-интерфейс рефлексии были спроектированы с учетом всех языков .NET; рефлексию можно использовать даже из кода Visual Basic.

Определенные конструкции языка C# – в частности, индексаторы, перечисления, операции и финализаторы – обрабатываются средой CLR особым образом.

- Индексатор C# транслируется в свойство, принимающее один или более аргументов, которое помечено как [DefaultMember] на уровне типа.
- Перечисление C# транслируется в подтип System.Enum со статическим полем для каждого члена.
- Операция C# транслируется в статический метод со специальным именем, начинающимся с op_; примером может служить op_Addition.
- Финализатор C# транслируется в метод, который переопределяет Finalize.

Еще одна сложность связана с тем, что свойства и события на самом деле заключают в себе два компонента:

- метаданные, описывающие свойство или событие (инкапсулированные посредством PropertyInfo или EventInfo);
- один или два поддерживающих метода.

В программе C# поддерживающие методы инкапсулированы внутри определения свойства или события. Но после компиляции в IL поддерживающие методы представляются как обычные методы, которые можно вызывать подобно любым другим. Другими словами, GetMethods наряду с обычными методами возвращает поддерживающие методы свойств и событий. Рассмотрим пример:

```
class Test { public int X { get { return 0; } set {} } }
void Demo()
{
  foreach (MethodInfo mi in typeof (Test).GetMethods())
    Console.Write (mi.Name + "  ");
}
// ВЫВОД:
get_X  set_X  GetType  ToString  Equals  GetHashCode
```

Идентифицировать эти методы можно через свойство IsSpecialName в классе MethodInfo. Свойство IsSpecialName возвращает true для методов доступа к свойствам, индексаторам и событиям, а также для операций. Оно возвращает false только для обычных методов C# и для метода Finalize, если определен финализатор.

Ниже представлены поддерживающие методы, генерируемые C#.

| Конструкция C# | Тип члена | Методы в IL |
|---|---|---|
| Свойство | Property | get_*XXX* и set_*XXX* |
| Индексатор | Property | get_Item и set_Item |
| Событие | Event | add_*XXX* и remove_*XXX* |

Каждый поддерживающий метод имеет собственный ассоциированный с ним объект MethodInfo. Получить к нему доступ можно следующим образом:

```
PropertyInfo pi = typeof (Console).GetProperty ("Title");
MethodInfo getter = pi.GetGetMethod(); // get_Title
MethodInfo setter = pi.GetSetMethod(); // set_Title
MethodInfo[] both = pi.GetAccessors(); // Length==2
```

Методы GetAddMethod и GetRemoveMethod делают аналогичную работу для класса EventInfo.

Чтобы двигаться в обратном направлении — из MethodInfo в связанный объект PropertyInfo или EventInfo — необходимо выполнять запрос. Для такой цели идеально подходит LINQ:

```
PropertyInfo p = mi.DeclaringType.GetProperties()
                  .First (x => x.GetAccessors (true).Contains (mi));
```

## Члены обобщенных типов

Метаданные членов можно получать как для несвязанных, так и для закрытых обобщенных типов:

```
PropertyInfo unbound = typeof (IEnumerator<>)  .GetProperty ("Current");
PropertyInfo closed = typeof (IEnumerator<int>).GetProperty ("Current");

Console.WriteLine (unbound);  // T Current
Console.WriteLine (closed);   // Int32 Current
Console.WriteLine (unbound.PropertyType.IsGenericParameter);  // True
Console.WriteLine (closed.PropertyType.IsGenericParameter);   // False
```

Объекты MemberInfo, возвращаемые из несвязанных и закрытых обобщенных типов, всегда отличаются — даже для членов, сигнатуры которых не содержат параметров обобщенных типов:

```
PropertyInfo unbound = typeof (List<>)  .GetProperty ("Count");
PropertyInfo closed = typeof (List<int>).GetProperty ("Count");

Console.WriteLine (unbound);  // Int32 Count
Console.WriteLine (closed);   // Int32 Count
Console.WriteLine (unbound == closed);  // False

Console.WriteLine (unbound.DeclaringType.IsGenericTypeDefinition); // True
Console.WriteLine (closed.DeclaringType.IsGenericTypeDefinition);  // False
```

Члены несвязанных обобщенных типов не могут *вызываться динамически.*

## Динамический вызов члена

Имея объект MethodInfo, PropertyInfo или FieldInfo, к нему можно динамически обращаться либо извлекать/устанавливать его значение. Это называется *динамическим связыванием* или *поздним связыванием*, т.к. выбор вызываемого члена производится во время выполнения, а не на этапе компиляции.

Например, в следующем коде применяется обычное *статическое связывание*:

```
string s = "Hello";
int length = s.Length;
```

А вот как то же самое сделать динамически с помощью рефлексии:

```
object s = "Hello";
PropertyInfo prop = s.GetType().GetProperty ("Length");
int length = (int) prop.GetValue (s, null);  // 5
```

Методы GetValue и SetValue извлекают и устанавливают значение объекта PropertyInfo или FieldInfo. Первый аргумент – экземпляр, который может быть null для статического члена. Доступ к индексатору подобен доступу к свойству по имени Item за исключением того, что при вызове метода GetValue или SetValue во втором аргументе указываются значения индексатора.

Чтобы вызвать метод динамически, необходимо обратиться к методу Invoke на объекте MethodInfo, предоставив массив аргументов, которые должны быть переданы вызываемому методу. Если окажется, что хотя бы один из аргументов имеет неподходящий тип, тогда во время выполнения сгенерируется исключение. При динамическом вызове утрачивается безопасность типов этапа компиляции, но по-прежнему поддерживается безопасность типов времени выполнения (такая же, как в случае использования ключевого слова dynamic).

## Параметры методов

Предположим, что необходимо динамически вызвать метод Substring типа string. Статически пришлось бы поступить следующим образом:

```
Console.WriteLine ("stamp".Substring(2));  // "amp"
```

Ниже показан динамический эквивалент, в котором применяется рефлексия:

```
Type type = typeof (string);
Type[] parameterTypes = { typeof (int) };
MethodInfo method = type.GetMethod ("Substring", parameterTypes);

object[] arguments = { 2 };
object returnValue = method.Invoke ("stamp", arguments);
Console.WriteLine (returnValue);  // "amp"
```

Поскольку метод Substring перегружен, мы должны передать методу GetMethod массив типов параметров, чтобы указать желаемую версию. Без типов параметров метод GetMethod сгенерирует исключение AmbiguousMatchException.

Метод GetParameters, определенный в MethodBase (базовый класс для MethodInfo и ConstructorInfo), возвращает метаданные параметров. Предыдущий пример можно продолжить:

```
ParameterInfo[] paramList = method.GetParameters();
foreach (ParameterInfo x in paramList)
{
  Console.WriteLine (x.Name);                // startIndex
  Console.WriteLine (x.ParameterType);       // System.Int32
}
```

### Работа с параметрами ref и out

Чтобы передать параметры ref или out, перед получением объекта метода необходимо вызвать MakeByRefType на типе. Например, представленный далее код:

```
int x;
bool successfulParse = int.TryParse ("23", out x);
```

можно выполнить динамически следующим образом:

```
object[] args = { "23", 0 };
Type[] argTypes = { typeof (string), typeof (int).MakeByRefType() };
MethodInfo tryParse = typeof (int).GetMethod ("TryParse", argTypes);
bool successfulParse = (bool) tryParse.Invoke (null, args);
Console.WriteLine (successfulParse + " " + args[1]);      // True 23
```

Тот же самый подход работает для типов параметров ref и out.

### Извлечение и вызов обобщенных методов

Явное указание типов параметров при вызове метода GetMethod может оказаться жизненно важным в разрешении неоднозначности перегруженных методов. Тем не менее, указывать типы обобщенных параметров невозможно. Например, рассмотрим класс System.Linq.Enumerable, в котором метод Where перегружен:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource, bool> predicate);
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);
```

Чтобы получить конкретную перегруженную версию, потребуется извлечь все методы и затем вручную найти желаемую версию. Приведенный далее запрос извлекает первую перегруженную версию метода Where:

```
from m in typeof (Enumerable).GetMethods()
where m.Name == "Where" && m.IsGenericMethod
let parameters = m.GetParameters()
where parameters.Length == 2
let genArg = m.GetGenericArguments().First()
let enumerableOfT = typeof (IEnumerable<>).MakeGenericType (genArg)
let funcOfTBool = typeof (Func<,>).MakeGenericType (genArg, typeof (bool))
where parameters[0].ParameterType == enumerableOfT
   && parameters[1].ParameterType == funcOfTBool
select m
```

Вызов .Single() здесь дает корректный объект MethodInfo с параметрами несвязанного типа. Следующий шаг предусматривает закрытие параметров типа посредством вызова метода MakeGenericMethod:

```
var closedMethod = unboundMethod.MakeGenericMethod (typeof (int));
```

В данном случае мы закрываем TSource с использованием int, что позволяет вызывать метод Enumerable.Where с source типа IEnumerable<int> и predicate типа Func<int,bool>:

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1;  // Только нечетные числа
```

Теперь закрытый обобщенный метод можно вызывать так:

```
var query = (IEnumerable<int>) closedMethod.Invoke
  (null, new object[] { source, predicate });
foreach (int element in query) Console.Write (element + "|");  // 3|5|7|
```

В случае применения API-интерфейса System.Linq.Expressions для динамического построения выражений (глава 8) беспокоиться по поводу указания обобщенного метода не придется. Метод Expression.Call перегружен, чтобы позволить указывать аргументы закрытого типа метода, который требуется вызвать:

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1;
var sourceExpr = Expression.Constant (source);
var predicateExpr = Expression.Constant (predicate);
var callExpression = Expression.Call (
  typeof (Enumerable), "Where",
  new[] { typeof (int) },  // Закрытый обобщенный тип аргумента.
  sourceExpr, predicateExpr);
```

## Использование делегатов для повышения производительности

Динамические вызовы относительно неэффективны и характеризуются накладными расходами, которые обычно укладываются в диапазон из нескольких микросекунд. Если метод вызывается многократно в цикле, то накладные расходы, приходящиеся на вызов, можно сместить в наносекундный диапазон, обращаясь вместо метода к динамически созданному экземпляру делегата, который нацелен на необходимый динамический метод. В следующем примере мы динамически вызываем метод Trim типа string миллион раз без значительных накладных расходов:

```
delegate string StringToString (string s);
static void Main()
{
  MethodInfo trimMethod = typeof (string).GetMethod ("Trim", new Type[0]);
  var trim = (StringToString) Delegate.CreateDelegate
                              (typeof (StringToString), trimMethod);
  for (int i = 0; i < 1000000; i++)
    trim ("test");
}
```

Такой код работает быстрее, потому что затратное динамическое связывание (код, выделенный полужирным) происходит только один раз.

## Доступ к неоткрытым членам

Все методы типов, применяемых для зондирования метаданных (например, `GetProperty`, `GetField` и т.д.), имеют перегруженные версии, которые принимают перечисление `BindingFlags`. Это перечисление служит фильтром метаданных и позволяет изменять стандартный критерий поиска. Наиболее распространенное использование связано с извлечением неоткрытых членов (работает только в настольных приложениях).

Например, пусть имеется следующий класс:

```
class Walnut
{
  private bool cracked;
  public void Crack() { cracked = true; }
  public override string ToString() { return cracked.ToString(); }
}
```

Вот как с ним можно поступить:

```
Type t = typeof (Walnut);
Walnut w = new Walnut();
w.Crack();
FieldInfo f = t.GetField ("cracked", BindingFlags.NonPublic |
                                     BindingFlags.Instance);
f.SetValue (w, false);
Console.WriteLine (w);  // False
```

Применение рефлексии для доступа к неоткрытым членам является мощной возможностью, однако оно также и небезопасно, поскольку позволяет обойти инкапсуляцию, создавая неуправляемую зависимость от внутренней реализации типа.

### Перечисление `BindingFlags`

Перечисление `BindingFlags` предназначено для побитового комбинирования. Чтобы получить любое совпадение, необходимо начать с одной из следующих четырех комбинаций:

```
BindingFlags.Public    | BindingFlags.Instance
BindingFlags.Public    | BindingFlags.Static
BindingFlags.NonPublic | BindingFlags.Instance
BindingFlags.NonPublic | BindingFlags.Static
```

Флаг `NonPublic` охватывает квалификаторы доступа `internal`, `protected`, `protected internal` и `private`.

Приведенный ниже код извлекает все открытые статические члены типа `object`:

```
BindingFlags publicStatic = BindingFlags.Public | BindingFlags.Static;
MemberInfo[] members = typeof (object).GetMembers (publicStatic);
```

В показанном далее примере извлекаются все неоткрытые члены типа `object`, как статические, так и экземпляра:

```
BindingFlags nonPublicBinding =
  BindingFlags.NonPublic | BindingFlags.Static | BindingFlags.Instance;
MemberInfo[] members = typeof (object).GetMembers (nonPublicBinding);
```

Флаг `DeclaredOnly` исключает функции, унаследованные от базовых типов, если только они не были переопределены.

Флаг DeclaredOnly может несколько запутывать тем, что он *ограничивает* результирующий набор (тогда как все остальные флаги результирующий набор *расширяют*).

## Обобщенные методы

Обобщенные методы не могут вызываться напрямую; следующий код приведет к генерации исключения:

```
class Program
{
  public static T Echo<T> (T x) { return x; }

  static void Main()
  {
    MethodInfo echo = typeof (Program).GetMethod ("Echo");
    Console.WriteLine (echo.IsGenericMethodDefinition);  // True
    echo.Invoke (null, new object[] { 123 } );     // Генерируется исключение
  }
}
```

Здесь потребуется дополнительный шаг, который предусматривает вызов метода MakeGenericMethod на объекте MethodInfo с указанием конкретных значений для аргументов обобщенных типов. В результате возвращается другой объект MethodInfo, к которому можно затем обращаться, как показано ниже:

```
MethodInfo echo = typeof (Program).GetMethod ("Echo");
MethodInfo intEcho = echo.MakeGenericMethod (typeof (int));
Console.WriteLine (intEcho.IsGenericMethodDefinition);          // False
Console.WriteLine (intEcho.Invoke (null, new object[] { 3 } )); // 3
```

## Анонимный вызов членов обобщенного интерфейса

Рефлексия удобна, когда необходимо вызвать член обобщенного интерфейса, а параметры типа не известны вплоть до времени выполнения. Теоретически если типы спроектированы идеально, то потребность в подобном действии возникает редко; тем не менее, естественно, типы далеко не всегда проектируются идеальным образом.

Например, предположим, что нужно написать более мощную версию метода ToString, которая могла бы развертывать результат выполнения запросов LINQ. Мы могли бы начать так:

```
public static string ToStringEx <T> (IEn     umerable<T> sequence)
{
  ...
}
```

Это уже довольно ограничено. Что если параметр sequence содержит вложенные коллекции, по которым также необходимо выполнить перечисление? Чтобы справиться с такой задачей, приведенный метод придется перегрузить:

```
public static string ToStringEx <T> (IEnumerable<IEnumerable<T>> sequence)
```

А если sequence содержит группы или *проекции* вложенных последовательностей? Статическое решение перегрузки методов становится непрактичным – нам необходим подход, который допускает масштабирование с целью обработки произвольного графа объектов вроде такого:

```
public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  StringBuilder sb = new StringBuilder();
  if (value is List<>)                                          // Ошибка
    sb.Append ("List of " + ((List<>) value).Count + " items"); // Ошибка
  if (value is IGrouping<,>)                                    // Ошибка
    sb.Append ("Group with key=" + ((IGrouping<,>) value).Key); // Ошибка
  // Выполнить перечисление элементов коллекции, если это коллекция,
  // рекурсивно вызывая метод ToStringEx
  // ...
  return sb.ToString();
}
```

К сожалению, код не скомпилируется: обращаться к членам *несвязанного* обобщенного типа, такого как `List<>` или `IGrouping<>`, нельзя. В случае `List<>` проблему можно решить за счет использования вместо него необобщенного интерфейса `IList`:

```
if (value is IList)
  sb.AppendLine ("A list with " + ((IList) value).Count + " items");
```

Так можно поступать потому, что проектировщики типа `List<>` предусмотрительно реализовали классический интерфейс `IList` (а также *обобщенный* интерфейс `IList`). Тот же самый принцип полезно принимать во внимание при написании собственных обобщенных типов: наличие необобщенного интерфейса или базового класса, к которому потребители смогут прибегнуть как к запасному варианту, может оказаться исключительно полезным.

Для `IGrouping<,>` решение не настолько простое. Интерфейс `IGrouping<,>` определен следующим образом:

```
public interface IGrouping <TKey,TElement> : IEnumerable <TElement>,
                                             IEnumerable
{
  TKey Key { get; }
}
```

Здесь нет никакого необобщенного типа, который можно было бы применить для доступа к свойству `Key`, поэтому в данном случае придется использовать рефлексию. Решение заключается в том, чтобы обращаться не к членам несвязанного обобщенного типа (что невозможно), а к членам *закрытого* обобщенного типа, чьи аргументы типа устанавливаются во время выполнения.

В следующей главе мы решим такую задачу более простым способом с помощью ключевого слова `dynamic` языка C#. Хорошим признаком для применения динамического связывания является ситуация, когда иначе приходится предпринимать разнообразные трюки с типами, как делается в настоящий момент.

На первом шаге понадобится выяснить, реализует ли `value` интерфейс `IGrouping<,>`, и если да, то получить закрытый обобщенный интерфейс. Проще всего это сделать с помощью запроса LINQ. Затем производится извлечение и обращение к свойству `Key`:

```
public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  if (value.GetType().IsPrimitive) return value.ToString();

  StringBuilder sb = new StringBuilder();

  if (value is IList)
    sb.Append ("List of " + ((IList)value).Count + " items: ");

  Type closedIGrouping = value.GetType().GetInterfaces()
    .Where (t => t.IsGenericType &&
                 t.GetGenericTypeDefinition() == typeof (IGrouping<,>))
    .FirstOrDefault();

  if (closedIGrouping != null) //Обратиться к свойству Key реализации IGrouping<,>
  {
    PropertyInfo pi = closedIGrouping.GetProperty ("Key");
    object key = pi.GetValue (value, null);
    sb.Append ("Group with key=" + key + ": ");
  }

  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append (ToStringEx (element) + " ");

  if (sb.Length == 0) sb.Append (value.ToString());

  return "\r\n" + sb.ToString();
}
```

Такой подход надежен: он работает независимо от того, как реализован интерфейс `IGrouping<,>` — неявно или явно. В следующем коде демонстрируется использование метода `ToStringEx`:

```
Console.WriteLine (ToStringEx (new List<int> { 5, 6, 7 } ));
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c) ));
```

```
List of 3 items: 5 6 7

Group with key=x: x
Group with key=y: y y
Group with key=z: z z z
```

# Рефлексия сборок

Для выполнения рефлексии сборки динамическим образом понадобится вызвать метод `GetType` или `GetTypes` на объекте `Assembly`. Приведенный ниже код извлекает из текущей сборки тип по имени `TestProgram`, определенный в пространстве имен `Demos`:

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

В приложении Windows Store сборку можно получить из существующего типа:

```
typeof (Foo).GetTypeInfo().Assembly.GetType ("Demos.TestProgram");
```

В следующем примере выводится список всех типов из сборки `mylib.dll`, находящейся в каталоге `e:\demo`:

```
Assembly a = Assembly.LoadFrom (@"e:\demo\mylib.dll");

foreach (Type t in a.GetTypes())
  Console.WriteLine (t);
```

Или:

```
Assembly a = typeof (Foo).GetTypeInfo().Assembly;

foreach (Type t in a.ExportedTypes)
  Console.WriteLine (t);
```

Метод GetTypes и свойство ExportedTypes возвращают только типы верхнего уровня, но не вложенные типы.

## Загрузка сборки в контекст, предназначенный только для рефлексии

В предыдущем примере для вывода списка типов сборки мы загружали ее в текущий домен приложения. В результате могут возникать нежелательные побочные эффекты, такие как выполнение статических конструкторов или нарушение последующего распознавания типов. Если нужно только проинспектировать информацию о типах (не создавая экземпляров и не обращаясь к их членам), то решением будет загрузка сборки в контекст, *предназначенный только для рефлексии* (допускается только в настольных приложениях):

```
Assembly a = Assembly.ReflectionOnlyLoadFrom (@"e:\demo\mylib.dll");
Console.WriteLine (a.ReflectionOnly);  // True

foreach (Type t in a.GetTypes())
  Console.WriteLine (t);
```

Это стартовая точка для написания браузера классов.

Для загрузки сборки в контекст, предназначенный только для рефлексии, предусмотрены три метода:

- ReflectionOnlyLoad (byte[])
- ReflectionOnlyLoad (string)
- ReflectionOnlyLoadFrom (string)

Даже в контексте, предназначенном только для рефлексии, загрузка множества версий сборки mscorlib.dll невозможна. Обходной прием предусматривает применение библиотек CCI от Microsoft (http://archive.codeplex.com/?p=cciast) или Mono.Cecil (http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/).

## Модули

Вызов метода GetTypes на многомодульной сборке возвращает все типы из всех модулей. В результате существование модулей можно проигнорировать и трактовать сборку как контейнер для типов. Однако существует один случай, когда модули имеют значение – работа с маркерами метаданных.

Маркер метаданных представляет собой целое число, которое уникальным образом ссылается на тип, член, строку или ресурс внутри области видимости модуля. Язык IL использует маркеры метаданных, а потому при синтаксическом разборе кода IL вы должны иметь возможность их распознавания. Предназначенные для такой цели методы определены в типе Module и называются ResolveType, ResolveMember, ResolveString и ResolveSignature. Мы вернемся к ним в последнем разделе главы при написании дизассемблера.

Получить список всех модулей в сборке можно с помощью метода `GetModules`. Свойство `ManifestModule` позволяет напрямую обращаться к главному модулю сборки.

# Работа с атрибутами

Среда CLR позволяет посредством атрибутов присоединять дополнительные метаданные к типам, членам и сборкам. Это механизм, с помощью которого производится управление многими функциями CLR, такими как сериализация и безопасность, что делает атрибуты неотъемлемой частью приложения.

Ключевая характеристика механизма атрибутов заключается в том, что можно создавать собственные атрибуты и затем применять их подобно любым другим атрибутам для "декорирования" элементов кода дополнительной информацией. Такая дополнительная информация компилируется внутрь лежащей в основе сборки и может быть извлечена во время выполнения с использованием рефлексии для построения работающих декларативно служб, подобных автоматизированному модульному тестированию.

## Основы атрибутов

Существуют три вида атрибутов:

- атрибуты с побитовым отображением;
- специальные атрибуты;
- псевдоспециальные атрибуты.

Из них только *специальные атрибуты* являются расширяемыми.

Сам по себе термин "атрибуты" может относиться к любому из указанных выше трех разновидностей, хотя в мире C# чаще всего будут иметься в виду специальные или псевдоспециальные атрибуты.

Атрибуты с побитовым отображением (наш термин) отображаются на выделенные биты в метаданных типа. Большинство ключевых слов модификаторов C# вроде `public`, `abstract` и `sealed` компилируются именно в атрибуты с побитовым отображением. Эти атрибуты очень эффективны, т.к. они задействуют минимальное пространство в метаданных (обычно всего лишь один бит), и среда CLR может находить их с небольшими затратами или вообще без таковых. Доступ к ним в API-интерфейсе рефлексии открывается через выделенные свойства класса `Type` (и других подклассов `MemberInfo`), такие как `IsPublic`, `IsAbstract` и `IsSealed`. Свойство `Attributes` возвращает перечисление флагов, которое описывает большинство атрибутов:

```
static void Main()
{
  TypeAttributes ta = typeof (Console).Attributes;
  MethodAttributes ma = MethodInfo.GetCurrentMethod().Attributes;
  Console.WriteLine (ta + "\r\n" + ma);
}
```

Ниже показан результат:

```
AutoLayout, AnsiClass, Class, Public, Abstract, Sealed, BeforeFieldInit
PrivateScope, Private, Static, HideBySig
```

По контрасту *специальные атрибуты* компилируются в двоичный блок, который находится в главной таблице метаданных типа. Все специальные атрибуты представлены подклассом класса `System.Attribute` и в отличие от атрибутов с побитовым отображением являются расширяемыми. Двоичный блок в метаданных идентифицирует класс атрибута, а также хранит значения любых позиционных либо именованных аргументов, которые были указаны, когда атрибут применялся. Специальные атрибуты, определяемые вами самостоятельно, архитектурно идентичны атрибутам, которые определены инфраструктурой .NET Framework.

В главе 4 было показано, каким образом присоединять специальные атрибуты к типу или члену в C#. Вот как присоединить предопределенный атрибут `Obsolete` к классу `Foo`:

```
[Obsolete] public class Foo {...}
```

Это инструктирует компилятор о необходимости встраивания в метаданные для `Foo` экземпляра `ObsoleteAttribute`, который затем может быть извлечен через рефлексию во время выполнения вызовом метода `GetCustomAttributes` на объекте `Type` или `MemberInfo`.

*Псевдоспециальные атрибуты* выглядят и ведут себя подобно стандартным специальным атрибутам. Они представлены подклассом класса `System.Attribute` и присоединяются в стандартной манере:

```
[Serializable] public class Foo {...}
```

Отличие в том, что компилятор или среда CLR внутреннее оптимизирует псевдоспециальные атрибуты, преобразуя их в атрибуты с побитовым отображением. Примеры включают `[Serializable]` (глава 17), `StructLayout`, `In` и `Out` (глава 25). Рефлексия открывает доступ к псевдоспециальным атрибутам через выделенные свойства вроде `IsSerializable`, и во многих случаях они также возвращаются в виде объектов `System.Attribute` при вызове метода `GetCustomAttributes` (в том числе `SerializableAttribute`). Это означает, что разница между псевдоспециальными и специальными атрибутами может быть (практически) проигнорирована (заметное исключение – использование пространства имен `Reflection.Emit` для динамической генерации типов во время выполнения; данная тема будет раскрыта в разделе "Выпуск сборок и типов" далее в главе).

## Атрибут `AttributeUsage`

`AttributeUsage` является атрибутом, применяемым к классам атрибутов. Он сообщает компилятору о том, как должен использоваться целевой атрибут:

```
public sealed class AttributeUsageAttribute : Attribute
{
  public AttributeUsageAttribute (AttributeTargets validOn);
  public bool AllowMultiple        { get; set; }
  public bool Inherited            { get; set; }
  public AttributeTargets ValidOn  { get; }
}
```

Свойство `AllowMultiple` управляет тем, может ли определяемый атрибут применяться к одной и той же цели более одного раза. Свойство `Inherited` указывает на то, должен ли атрибут, примененный к базовому классу, также применяться к производным классам (или в случае методов – должен ли атрибут, примененный к виртуальному методу, также применяться к переопределенным методам). Свойство `ValidOn` определяет набор целей (классов, интерфейсов, свойств, методов, параметров и т.д.),

к которым может быть присоединен атрибут. Оно принимает любую комбинацию значений перечисления AttributeTargets, которое содержит следующие члены:

```
All                    GenericParameter
Assembly               Interface
Class                  Method
Constructor            Module
Delegate               Parameter
Enum                   Property
Event                  ReturnValue
Field                  Struct
```

В целях иллюстрации ниже показано, как авторы инфраструктуры .NET Framework применили атрибут AttributeUsage к атрибуту Serializable:

```
[AttributeUsage (AttributeTargets.Delegate |
                 AttributeTargets.Enum     |
                 AttributeTargets.Struct   |
                 AttributeTargets.Class,    Inherited = false)
]
public sealed class SerializableAttribute : Attribute { }
```

Фактически это почти полное определение атрибута Serializable. Написание класса атрибута, не имеющего свойств или специальных конструкторов, столь же просто.

## Определение собственного атрибута

Ниже перечислены шаги, которые должны быть выполнены для определения собственного атрибута.

1. Создайте класс, производный от System.Attribute или от потомка System.Attribute. По соглашению имя класса должно заканчиваться словом Attribute, хотя поступать так не обязательно.
2. Примените атрибут AttributeUsage, описанный в предыдущем разделе.

   Если атрибут не требует каких-либо свойств или аргументов в своем конструкторе, то работа завершена.
3. Напишите один или более открытых конструкторов. Параметры конструктора определяют позиционные параметры атрибута и становятся обязательными при использовании атрибута.
4. Объявите открытое поле или свойство для каждого именованного параметра, который планируется поддерживать. При использовании атрибута именованные параметры необязательны.

Свойства атрибута и параметры конструктора должны относиться к следующим типам:

- запечатанный примитивный тип, т.е. bool, byte, char, double, float, int, long, short или string;
- тип Type;
- тип перечисления;
- одномерный массив любого из упомянутых выше типов.

Когда атрибут применяется, у компилятора также должна быть возможность статической оценки каждого свойства или аргумента конструктора.

В следующем классе определяется атрибут для содействия системе автоматизированного модульного тестирования. Он указывает, что метод должен быть протестирован, устанавливает количество повторений теста и задает сообщение, выдаваемое в случае неудачи:

```
[AttributeUsage (AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
  public int     Repetitions;
  public string  FailureMessage;

  public TestAttribute () : this (1)      { }
  public TestAttribute (int repetitions) { Repetitions = repetitions; }
}
```

Ниже представлен код класса `Foo` с методами, которые декорированы атрибутом `Test` разнообразными способами:

```
class Foo
{
  [Test]
  public void Method1() { ... }

  [Test(20)]
  public void Method2() { ... }

  [Test(20, FailureMessage="Debugging Time!")]
  public void Method3() { ... }
}
```

## Извлечение атрибутов во время выполнения

Есть два стандартных способа извлечения атрибутов во время выполнения:

- вызов метода `GetCustomAttributes` на любом объекте `Type` или `MemberInfo`;
- вызов метода `Attribute.GetCustomAttribute` или `Attribute.GetCustomAttributes`.

Последние два метода перегружены для приема любого объекта рефлексии, который соответствует допустимой цели атрибута (`Type`, `Assembly`, `Module`, `MemberInfo` или `ParameterInfo`).

Начиная с версии .NET Framework 4.0, для получения информации об атрибутах можно также вызвать метод `GetCustomAttributes`**`Data`** на типе или члене. Отличие между этим методом и `GetCustomAttributes` в том, что первый из них сообщает, *каким образом* атрибут создавался: он указывает перегруженную версию конструктора, которая была использована, и значение каждого аргумента и именованного параметра конструктора. Такие сведения полезны, когда требуется выпускать код или IL для воссоздания атрибута в том же самом состоянии (как объясняется в разделе "Выпуск членов типа" далее в главе).

Ниже показано, как можно выполнить перечисление всех методов в предшествующем классе `Foo`, которые имеют атрибут `TestAttribute`:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));
```

```
  if (att != null)
    Console.WriteLine ("Method {0} will be tested; reps={1}; msg={2}",
                       mi.Name, att.Repetitions, att.FailureMessage);
}
```

Или:

```
foreach (MethodInfo mi in typeof (Foo).GetTypeInfo().DeclaredMethods)
...
```

Вот вывод:

```
Method Method1 will be tested; reps=1; msg=
Method Method2 will be tested; reps=20; msg=
Method Method3 will be tested; reps=20; msg=Debugging Time!
```

Чтобы завершить демонстрацию применения таких приемов при написании системы модульного тестирования, ниже представлен тот же самый пример, расширенный так, чтобы на самом деле вызывать методы, декорированные атрибутом `[Test]`:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));
  if (att != null)
    for (int i = 0; i < att.Repetitions; i++)
      try
      {
        mi.Invoke (new Foo(), null);  // Вызвать метод без аргументов
      }
      catch (Exception ex)  // Поместить исключение внутрь att.FailureMessage
      {
        throw new Exception ("Error: " + att.FailureMessage, ex);  // Ошибка
      }
}
```

Возвращаясь к рефлексии атрибутов, далее показан пример, в котором выводится список атрибутов, присутствующих в заданном типе:

```
[Serializable, Obsolete]
class Test
{
  static void Main()
  {
    object[] atts = Attribute.GetCustomAttributes (typeof (Test));
    foreach (object att in atts) Console.WriteLine (att);
  }
}
```

Вывод будет следующим:

```
System.ObsoleteAttribute
System.SerializableAttribute
```

## Извлечение атрибутов в контексте, предназначенном только для рефлексии

Вызов метода `GetCustomAttributes` на члене, который загружен в контекст, предназначенный только для рефлексии, запрещен, потому что он требует создания

произвольно типизированных атрибутов (вспомните, что создание объектов в контексте, предназначенном только для рефлексии, не разрешено). Для обхода данного ограничения предусмотрен специальный тип по имени CustomAttributeData, который позволяет выполнять рефлексию таких атрибутов. Ниже приведен пример его использования:

```
IList<CustomAttributeData> atts = CustomAttributeData.GetCustomAttributes
                                  (myReflectionOnlyType);
foreach (CustomAttributeData att in atts)
{
  Console.Write (att.GetType());            // Тип атрибута
  Console.WriteLine (" " + att.Constructor);// Объект ConstructorInfo
  foreach (CustomAttributeTypedArgument arg in att.ConstructorArguments)
    Console.WriteLine ("  " +arg.ArgumentType + "=" + arg.Value);
  foreach (CustomAttributeNamedArgument arg in att.NamedArguments)
    Console.WriteLine ("  " + arg.MemberInfo.Name + "=" + arg.TypedValue);
}
```

Во многих случаях типы атрибутов будут находиться в другой сборке, отличающейся от той, для которой выполняется рефлексия. Один из способов справиться с этим предполагает обработку события ReflectionOnlyAssemblyResolve в текущем домене приложения:

```
ResolveEventHandler handler = (object sender, ResolveEventArgs args)
                              => Assembly.ReflectionOnlyLoad (args.Name);
AppDomain.CurrentDomain.ReflectionOnlyAssemblyResolve += handler;
// Рефлексия по атрибутам...
AppDomain.CurrentDomain.ReflectionOnlyAssemblyResolve -= handler;
```

# Динамическая генерация кода

Пространство имен System.Reflection.Emit содержит классы для создания метаданных и кода IL во время выполнения. Генерация кода динамическим образом полезна для решения определенных видов задач программирования. Примером может служить API-интерфейс регулярных выражений, который выпускает типы, настроенные на специфические регулярные выражения. Другие применения Reflection.Emit в .NET Framework включают динамическую генерацию прозрачных прокси для технологии Remoting и генерацию типов, которые выполняют специальные XSLT-преобразования с минимальными накладными расходами во время выполнения. Инструмент LINQPad использует пространство имен Reflection.Emit для динамической генерации типизированных классов DataContext.

В приложениях Windows Store и .NET Core пространство имен Reflection.Emit не поддерживается.

## Генерация кода IL с помощью класса DynamicMethod

Класс DynamicMethod — это легковесный инструмент в пространстве имен System.Reflection.Emit, предназначенный для генерации методов на лету. В отличие от TypeBuilder он не требует предварительной установки динамической сборки, модуля и типа, в котором должен содержаться метод. Такие характеристики делают класс DynamicMethod подходящим средством для решения простых задач, а также хорошим введением в пространство имен Reflection.Emit.

Объект DynamicMethod и связанный с ним код IL подвергаются сборке мусора, когда на них больше нет ссылок. Это значит, что динамические методы можно генерировать многократно, не заполняя излишне память. (Чтобы делать то же самое с динамическими *сборками*, при создании сборки потребуется применить флаг AssemblyBuilderAccess.RunAndCollect.)

Ниже представлен простой пример использования класса DynamicMethod для создания метода, который выводит на консоль строку Hello world:

```
public class Test
{
  static void Main()
  {
    var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator();
    gen.EmitWriteLine ("Hello world");
    gen.Emit (OpCodes.Ret);
    dynMeth.Invoke (null, null);  // Hello world
  }
}
```

Для каждого кода операции IL в классе OpCodes имеется статическое поле, допускающее только чтение. Большая часть функциональности доступна через различные коды операций, хотя в классе ILGenerator также есть специализированные методы для генерации меток и локальных переменных и для обработки исключений. Метод всегда завершается кодом операции Opcodes.Ret, который означает "возврат", или разновидностью инструкции ветвления/генерации. Метод EmitWriteLine класса ILGenerator — сокращение для выпуска нескольких кодов операций более низкого уровня. Мы могли бы заменить вызов EmitWriteLine, как показано далее, и получить тот же самый результат:

```
MethodInfo writeLineStr = typeof (Console).GetMethod ("WriteLine",
                          new Type[] { typeof (string) });
gen.Emit (OpCodes.Ldstr, "Hello world");  // Загрузить строку
gen.Emit (OpCodes.Call, writeLineStr);    // Вызвать метод
```

Обратите внимание, что мы передаем конструктору DynamicMethod аргумент typeof(Test). Это предоставляет динамическому методу доступ к неоткрытым методам данного типа, разрешая поступать следующим образом:

```
public class Test
{
  static void Main()
  {
    var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator();
    MethodInfo privateMethod = typeof(Test).GetMethod ("HelloWorld",
      BindingFlags.Static | BindingFlags.NonPublic);
    gen.Emit (OpCodes.Call, privateMethod);  // Вызвать метод HelloWorld
    gen.Emit (OpCodes.Ret);
    dynMeth.Invoke (null, null);  // Hello world
  }
  static void HelloWorld()        // Закрытый метод, но мы можем вызвать его
  {
    Console.WriteLine ("Hello world");
  }
}
```

Освоение языка IL требует существенного времени. Вместо запоминания всех кодов операций намного проще скомпилировать какую-нибудь программу C# и затем исследовать, копировать и настраивать код IL. Средство LINQPad отображает код IL для любого метода или фрагмента кода, который вы введете, а инструменты для просмотра сборок, такие как `ildasm` или .NET Reflector, удобны для изучения существующих сборок.

## Стек оценки

Центральной концепцией в IL является *стек оценки* (evaluation stack). Чтобы вызвать метод с аргументами, сначала понадобится затолкнуть ("загрузить") аргументы в стек оценки и затем вызвать метод. Впоследствии метод извлекает необходимые аргументы из стека оценки. Мы демонстрировали прием ранее при вызове `Console.WriteLine`. Ниже приведен похожий пример с целым числом:

```
var dynMeth = new DynamicMethod ("Foo", null, null, typeof(void));
ILGenerator gen = dynMeth.GetILGenerator();
MethodInfo writeLineInt = typeof (Console).GetMethod ("WriteLine",
                                                      new Type[] { typeof (int) });

// Коды операций Ldc* загружают числовые литералы различных типов и размеров.

gen.Emit (OpCodes.Ldc_I4, 123);  // Затолкнуть в стек 4-байтовое целое число
gen.Emit (OpCodes.Call, writeLineInt);

gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);     // 123
```

Чтобы сложить два числа, нужно загрузить их в стек оценки и вызвать `Add`. Код операции `Add` извлекает два значения из стека оценки и заталкивает результат обратно в стек. Следующий код суммирует числа 2 и 2, после чего выводит результат с применением полученного ранее метода `WriteLine`:

```
gen.Emit (OpCodes.Ldc_I4, 2); //Затолкнуть 4-байтовое целое число, значение = 2
gen.Emit (OpCodes.Ldc_I4, 2); //Затолкнуть 4-байтовое целое число, значение = 2
gen.Emit (OpCodes.Add);       // Сложить и получить результат
gen.Emit (OpCodes.Call, writeLineInt);
```

Чтобы вычислить выражение `10 / 2 + 1`, можно поступить либо так:

```
gen.Emit (OpCodes.Ldc_I4, 10);
gen.Emit (OpCodes.Ldc_I4, 2);
gen.Emit (OpCodes.Div);
gen.Emit (OpCodes.Ldc_I4, 1);
gen.Emit (OpCodes.Add);
gen.Emit (OpCodes.Call, writeLineInt);
```

либо так:

```
gen.Emit (OpCodes.Ldc_I4, 1);
gen.Emit (OpCodes.Ldc_I4, 10);
gen.Emit (OpCodes.Ldc_I4, 2);
gen.Emit (OpCodes.Div);
gen.Emit (OpCodes.Add);
gen.Emit (OpCodes.Call, writeLineInt);
```

## Передача аргументов динамическому методу

Загрузить в стек аргумент, переданный динамическому методу, можно с помощью кодов операций Ldarg и Ldarg_*XXX*. Чтобы значение возвратилось, перед завершением оно должно оставаться единственным значением в стеке. Для этого при вызове конструктора DynamicMethod потребуется указать возвращаемый тип и типы аргументов. В показанном ниже коде создается динамический метод, который возвращает сумму двух целых чисел:

```
DynamicMethod dynMeth = new DynamicMethod ("Foo",
  typeof (int),                          // Возвращаемый тип: int
  new[] { typeof (int), typeof (int) }, // Типы параметров: int, int
  typeof (void));

ILGenerator gen = dynMeth.GetILGenerator();

gen.Emit (OpCodes.Ldarg_0)  // Затолкнуть в стек оценки первый аргумент
gen.Emit (OpCodes.Ldarg_1); // Затолкнуть в стек оценки второй аргумент
gen.Emit (OpCodes.Add);     // Сложить аргументы (результат остается в стеке)
gen.Emit (OpCodes.Ret);     // Возврат при стеке, содержащем одно значение

int result = (int) dynMeth.Invoke (null, new object[] { 3, 4 } );  // 7
```

По завершении стек оценки должен содержать в точности 0 или 1 элемент (в зависимости от того, возвращает ли метод значение). Если нарушить данное требование, то среда CLR откажется выполнять метод. Удалить элемент из стека без обработки можно с помощью кода операции OpCodes.Pop.

Вместо вызова Invoke иногда удобнее оперировать динамическим методом как типизированным делегатом, для чего предназначен метод CreateDelegate. В качестве примера предположим, что определен делегат по имени BinaryFunction:

```
delegate int BinaryFunction (int n1, int n2);
```

Тогда последнюю строку в предыдущем примере можно было бы заменить следующими строками:

```
BinaryFunction f = (BinaryFunction) dynMeth.CreateDelegate
                                    (typeof (BinaryFunction));
int result = f (3, 4);  // 7
```

Делегат также устраняет накладные расходы, связанные с динамическим вызовом метода, экономя несколько микросекунд на вызов.

Мы покажем, как передавать ссылку, в разделе "Выпуск членов типа" далее в главе.

## Генерация локальных переменных

Объявить локальную переменную можно путем вызова метода DeclareLocal на экземпляре ILGenerator. В результате возвращается объект LocalBuilder, который можно использовать в сочетании с кодами операций, такими как Ldloc (загрузить локальную переменную) или Stloc (сохранить локальную переменную). Ldloc заталкивает в стек оценки, а Stloc извлекает из него.

Например, взгляните на показанный далее код C#:

```
int x = 6;
int y = 7;
x *= y;
Console.WriteLine (x);
```

Приведенный ниже код динамически генерирует предыдущий код:

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();

LocalBuilder localX = gen.DeclareLocal (typeof (int));    // Объявить
                                                          // переменную x
LocalBuilder localY = gen.DeclareLocal (typeof (int));    // Объявить
                                                          // переменную y

gen.Emit (OpCodes.Ldc_I4, 6);      // Затолкнуть в стек оценки литерал 6
gen.Emit (OpCodes.Stloc, localX); // Сохранить в localX
gen.Emit (OpCodes.Ldc_I4, 7);      // Затолкнуть в стек оценки литерал 7
gen.Emit (OpCodes.Stloc, localY); // Сохранить в localY

gen.Emit (OpCodes.Ldloc, localX)  // Затолкнуть в стек оценки localX
gen.Emit (OpCodes.Ldloc, localY); // Затолкнуть в стек оценки localY
gen.Emit (OpCodes.Mul);           // Перемножить значения
gen.Emit (OpCodes.Stloc, localX); // Сохранить результат в localX

gen.EmitWriteLine (localX);       // Вывести значение localX
gen.Emit (OpCodes.Ret);

dynMeth.Invoke (null, null);      // 42
```

Инструмент .NET Reflector от Redgate великолепно подходит для исследования динамических методов на предмет ошибок: произведя декомпиляцию в код C#, обычно довольно легко выяснить, что было сделано не так! Мы объясним, как сохранять результаты динамической генерации на диске, в разделе "Выпуск сборок и типов" далее в главе. Другим удобным инструментом является визуализатор IL от Microsoft для Visual Studio (`http://albahari.com/ilvisualizer`).

## Ветвление

В языке IL отсутствуют циклы вроде `while`, `do` и `for`; вся работа делается с помощью меток плюс эквивалентов оператора `goto` и условного оператора `goto`. Существуют коды операций ветвления, такие как `Br` (безусловное ветвление), `Brtrue` (ветвление, если значение в стеке оценки равно `true`) и `Blt` (ветвление, если первое значение меньше второго значения).

Для установки цели ветвления сначала понадобится вызвать метод `DefineLabel` (он возвращает объект `Label`) и затем вызвать метод `MarkLabel` в месте, к которому должна быть прикреплена метка. Например, рассмотрим следующий код C#:

```
int x = 5;
while (x <= 10) Console.WriteLine (x++);
```

Выпустить его можно так:

```
ILGenerator gen = ...

Label startLoop = gen.DefineLabel();  // Объявить метки
Label endLoop = gen.DefineLabel();
```

```
LocalBuilder x = gen.DeclareLocal (typeof (int));     // int x
gen.Emit (OpCodes.Ldc_I4, 5);                         //
gen.Emit (OpCodes.Stloc, x);                          // x = 5
gen.MarkLabel (startLoop);
  gen.Emit (OpCodes.Ldc_I4, 10);        // Загрузить в стек оценки 10
  gen.Emit (OpCodes.Ldloc, x);          // Загрузить в стек оценки x
  gen.Emit (OpCodes.Blt, endLoop);      // if (x > 10) goto endLoop
  gen.EmitWriteLine (x);                // Console.WriteLine (x)
  gen.Emit (OpCodes.Ldloc, x);          // Загрузить в стек оценки x
  gen.Emit (OpCodes.Ldc_I4, 1);         // Загрузить в стек оценки 1
  gen.Emit (OpCodes.Add);               // Выполнить сложение
  gen.Emit (OpCodes.Stloc, x);          // Сохранить результат в x
  gen.Emit (OpCodes.Br, startLoop);     // Вернуться в начало цикла
gen.MarkLabel (endLoop);
gen.Emit (OpCodes.Ret);
```

## Создание объектов и вызов методов экземпляра

Эквивалентом операции new в языке IL является код операции Newobj, который обращается к конструктору и загружает созданный объект в стек оценки. Например, следующий код конструирует объект StringBuilder:

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();

ConstructorInfo ci = typeof (StringBuilder).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Newobj, ci);
```

После помещения объекта в стек оценки можно вызывать его методы экземпляра, применяя код операции Call или Callvirt. Расширяя рассматриваемый пример, мы запросим свойство MaxCapacity объекта StringBuilder путем вызова метода доступа get свойства и затем выведем результат:

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder)
                      .GetProperty ("MaxCapacity").GetGetMethod());

gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine",
                                  new[] { typeof (int) } ));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);  // 2147483647
```

Вот как эмулировать семантику вызовов C#:

- используйте код операции Call для обращения к статическим методам и методам экземпляра типов значения;
- применяйте код операции Callvirt для обращения к методам экземпляра ссылочных типов (независимо от того, объявлены они виртуальными или нет).

В данном примере мы использовали Callvirt на экземпляре StringBuilder, несмотря на то, что свойство MaxCapacity не является виртуальным. Это не приводит к ошибке, а просто выполняет невиртуальный вызов. Вызов методов экземпляра ссылочных типов с помощью Callvirt позволяет избежать риска возникновения противоположного условия: обращения к виртуальному методу посредством Call. (Риск вполне реален. Автор целевого метода может позже *изменить* его объявление.) Преимущество операции Callvirt также в том, что она обеспечивает проверку получателя на равенство null.

Вызов виртуального метода с помощью операции `Call` обходит семантику виртуальных вызовов и обращается к методу напрямую, что редко является желательным и на самом деле нарушает безопасность типов.

В следующем примере мы создаем объект `StringBuilder`, передавая конструктору два аргумента, добавляем к нему строку `", world!"` и вызываем метод `ToString` на этом объекте:

```
// Мы будем вызывать: new StringBuilder ("Hello", 1000)
ConstructorInfo ci = typeof (StringBuilder).GetConstructor (
                     new[] { typeof (string), typeof (int) } );
gen.Emit (OpCodes.Ldstr, "Hello"); // Загрузить в стек оценки строку
gen.Emit (OpCodes.Ldc_I4, 1000);   // Загрузить в стек оценки целое число
gen.Emit (OpCodes.Newobj, ci);     // Сконструировать объект StringBuilder
Type[] strT = { typeof (string) };
gen.Emit (OpCodes.Ldstr, ", world!");
gen.Emit (OpCodes.Call, typeof (StringBuilder).GetMethod ("Append", strT));
gen.Emit (OpCodes.Callvirt, typeof (object).GetMethod ("ToString"));
gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine", strT));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);       // Hello, world!
```

Ради интереса мы вызвали метод `GetMethod` на `typeof(object)`, после чего использовали операцию `Callvirt` для выполнения вызова виртуального метода на `ToString`. Тот же результат можно было бы получить, вызвав метод `ToString` на самом типе `StringBuilder`:

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder).GetMethod ("ToString",
                                                    new Type[0] ));
```

(При вызове методу `GetMethod` должен передаваться пустой массив `Type`, т.к. `StringBuilder` перегружает метод `ToString` с применением другой сигнатуры.)

Если бы метод `ToString` типа `object` вызывался невиртуальным образом:

```
gen.Emit (OpCodes.Call,
          typeof (object).GetMethod ("ToString"));
```

тогда результатом оказалась бы строка `"System.Text.StringBuilder"`. Другими словами, мы должны обойти версию `ToString`, переопределенную в классе `StringBuilder`, и вызвать версию данного метода из `object`.

## Обработка исключений

Класс `ILGenerator` предлагает выделенные методы для обработки исключений. Приведенный ниже код C#:

```
try                                { throw new NotSupportedException(); }
catch (NotSupportedException ex)   { Console.WriteLine (ex.Message);    }
finally                            { Console.WriteLine ("Finally");     }
```

можно сгенерировать следующим образом:

```
MethodInfo getMessageProp = typeof (NotSupportedException)
                            .GetProperty ("Message").GetGetMethod();
MethodInfo writeLineString = typeof (Console).GetMethod ("WriteLine",
                                 new[] { typeof (object) } );
gen.BeginExceptionBlock();
  ConstructorInfo ci = typeof (NotSupportedException).GetConstructor (
                                                      new Type[0] );
  gen.Emit (OpCodes.Newobj, ci);
  gen.Emit (OpCodes.Throw);
gen.BeginCatchBlock (typeof (NotSupportedException));
  gen.Emit (OpCodes.Callvirt, getMessageProp);
  gen.Emit (OpCodes.Call, writeLineString);
gen.BeginFinallyBlock();
  gen.EmitWriteLine ("Finally");
gen.EndExceptionBlock();
```

Как и в языке C#, можно иметь много блоков `catch`. Для повторной генерации исключения понадобится выпустить код операции `Rethrow`.

Класс `ILGenerator` предоставляет вспомогательный метод по имени `ThrowException`. Однако он содержит ошибку, которая не дает возможности его использовать с экземпляром `DynamicMethod`. Упомянутый метод работает только с экземпляром `MethodBuilder` (как будет показано в следующем разделе).

# Выпуск сборок и типов

Несмотря на удобство класса `DynamicMethod`, он может генерировать только методы. Если необходимо выпускать любые другие конструкции (или целый тип), то придется применять полный "тяжеловесный" API-интерфейс. Это означает динамическое построение сборки и модуля. Тем не менее, сборка не обязательно должна присутствовать на диске; она может располагаться полностью в памяти.

Давайте предположим, что требуется динамически построить тип. Поскольку тип должен находиться в модуле внутри сборки, необходимо сначала создать сборку и модуль, чтобы создание типа стало возможным. За такую работу отвечают классы `AssemblyBuilder` и `ModuleBuilder`:

```
AppDomain appDomain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName ("MyDynamicAssembly");
AssemblyBuilder assemBuilder =
  appDomain.DefineDynamicAssembly (aname, AssemblyBuilderAccess.Run);
ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule ("DynModule");
```

Добавить тип в существующую сборку не удастся, т.к. после создания сборка становится неизменяемой.

Динамические сборки не подвержены обработке сборщиком мусора и остаются в памяти вплоть до завершения домена приложения, если только при их определении не был указан флаг `AssemblyBuilderAccess.RunAndCollect`. К сборкам, которые могут быть обработаны сборщиком мусора, применяются различные ограничения (`https://msdn.microsoft.com/ru-ru/library/dd554932(v=vs.100).aspx`).

При наличии модуля, где может находиться тип, для создания типа можно использовать класс TypeBuilder. Вот как определить класс по имени Widget:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
```

Перечисление флагов TypeAttributes поддерживает модификаторы типов CLR, которые можно увидеть после дизассемблирования типа с помощью ildasm. Помимо флагов видимости членов это перечисление включает такие модификаторы типов, как Abstract и Sealed, а также Interface для определения интерфейса .NET. Кроме того, имеется флаг Serializable, который эквивалентен применению атрибута [Serializable] в C#, и Explicit, эквивалентный применению атрибута [StructLayout(LayoutKind.Explicit)]. Мы покажем, как работать с другими разновидностями атрибутов, в разделе "Присоединение атрибутов" далее в главе.

Метод DefineType также принимает необязательный базовый тип:

- для определения структуры укажите базовый тип System.ValueType;
- для определения делегата укажите базовый тип System.MulticastDelegate;
- для реализации интерфейса используйте конструктор, который принимает массив типов интерфейсов;
- для определения интерфейса укажите комбинацию TypeAttributes.Interface | TypeAttributes.Abstract.

Определение типа делегата требует выполнения нескольких дополнительных шагов. Джоэль Побар объясняет, как это сделать, в статье "Creating delegate types via Reflection.Emit" ("Создание типов делегатов через Reflection.Emit") своего блога, доступного по адресу http://blogs.msdn.com/joelpob/.

Теперь можно создавать члены внутри типа:

```
MethodBuilder methBuilder = tb.DefineMethod ("SayHello",
                                              MethodAttributes.Public,
                                              null, null);
ILGenerator gen = methBuilder.GetILGenerator();
gen.EmitWriteLine ("Hello world");
gen.Emit (OpCodes.Ret);
```

Итак, все готово для создания типа, и вот как завершить его определение:

```
Type t = tb.CreateType();
```

После того, как тип создан, с помощью обычной рефлексии его можно инспектировать и производить динамическое связывание:

```
object o = Activator.CreateInstance (t);
t.GetMethod ("SayHello").Invoke (o, null);  // Hello world
```

## Сохранение сгенерированных сборок

Метод Save класса AssemblyBuilder записывает сгенерированную сборку в файл с указанным именем. Однако чтобы он заработал, потребуется предпринять два действия:

- при конструировании объекта AssemblyBuilder указать флаг Save или RunAndSave из перечисления AssemblyBuilderAccess;
- при конструировании объекта ModuleBuilder указать имя файла (которое должно совпадать с именем файла сборки, если только не создается многомодульная сборка).

Можно также дополнительно установить свойства объекта `AssemblyName`, такие как `Version` или `KeyPair` (для подписания сборки).

Например:

```
AppDomain domain = AppDomain.CurrentDomain;

AssemblyName aname = new AssemblyName ("MyEmissions");
aname.Version = new Version (2, 13, 0, 1);

AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
  "MainModule", "MyEmissions.dll");

// Создать типы, как это делалось ранее...
// ...

assemBuilder.Save ("MyEmissions.dll");
```

Данный код записывает сборку в файл внутри базового каталога приложения. Чтобы сохранить файл в другом местоположении, при конструировании объекта `AssemblyBuilder` должен быть предоставлен альтернативный каталог:

```
AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave, @"d:\assemblies" );
```

После сохранения в файле динамическая сборка становится обычной сборкой, похожей на любую другую. В программе можно статически ссылаться на только что построенную сборку и поступать так:

```
Widget w = new Widget();
w.SayHello();
```

## Объектная модель `Reflection.Emit`

На рис. 19.2 показаны основные типы в пространстве имен `System.Reflection.Emit`.

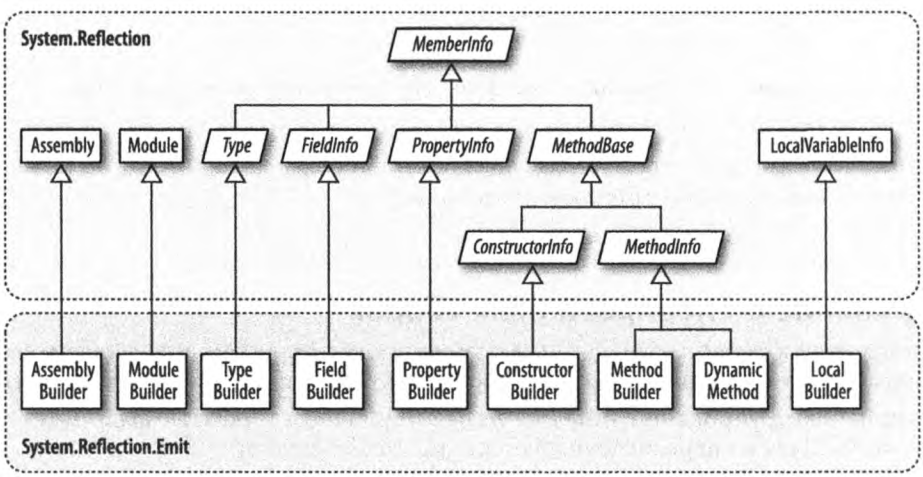



***Рис. 19.2. Пространство имен `System.Reflection.Emit`***

Каждый тип описывает конструкцию CLR и основан на эквиваленте из пространства System.Reflection. В результате при построении какого-то типа на месте обычных конструкций можно применять сгенерированные конструкции. Например, ранее мы вызывали метод Console.WriteLine следующим образом:

```
MethodInfo writeLine = typeof(Console).GetMethod ("WriteLine",
                                                  new Type[] { typeof (string) });
gen.Emit (OpCodes.Call, writeLine);
```

Мы могли бы столь же легко вызвать динамически сгенерированный метод, обратившись к методу gen.Emit и передав ему объект MethodBuilder, а не MethodInfo. Это очень важно – иначе не было бы возможности написать один динамический метод, который вызывает другой метод в том же типе.

Вспомните, что по завершении наполнения объекта TypeBuilder должен быть вызван его метод CreateType. Вызов CreateType запечатывает объект TypeBuilder и все его члены – так что ничего больше не может быть добавлено либо изменено – и возвращает обратно реальный тип Type, экземпляры которого можно создавать.

Перед вызовом метода CreateType объект TypeBuilder и его члены находятся в "несозданном" состоянии. Существуют значительные ограничения относительно того, что можно делать с несозданными конструкциями. В частности, нельзя вызывать члены, возвращающие объекты MemberInfo, такие как GetMembers, GetMethod или GetProperty – это приведет к генерации исключения. Чтобы сослаться на члены несозданного типа, придется использовать исходные выпуски:

```
TypeBuilder tb = ...

MethodBuilder method1 = tb.DefineMethod ("Method1", ...);
MethodBuilder method2 = tb.DefineMethod ("Method2", ...);

ILGenerator gen1 = method1.GetILGenerator();

// Предположим, что method1 должен вызывать method2:

gen1.Emit (OpCodes.Call, method2);                    // Правильно
gen1.Emit (OpCodes.Call, tb.GetMethod ("Method2"));  // Неправильно
```

После вызова метода CreateType можно проводить рефлексию и активизацию не только возвращенного объекта Type, но также исходного объекта TypeBuilder. Фактически TypeBuilder превращается в прокси для реального Type. Мы покажем, почему такая возможность важна, в разделе "Сложности, связанные с генерацией" далее в главе.

# Выпуск членов типа

Во всех примерах настоящего раздела предполагается, что объект типа TypeBuilder по имени tb был создан следующим образом:

```
AppDomain domain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName ("MyEmissions");

AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
  "MainModule", "MyEmissions.dll");

TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
```

## Выпуск методов

При вызове метода DefineMethod можно указывать возвращаемый тип и типы параметров в той же самой манере, как и при создании объекта DynamicMethod. Например, следующий метод:

```
public static double SquareRoot (double value)
{
  return Math.Sqrt (value);
}
```

может быть сгенерирован так:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  typeof (double),                    // Возвращаемый тип
  new[]  { typeof (double) } );       // Типы параметров

mb.DefineParameter (1, ParameterAttributes.None, "value"); // Назначить имя

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);           // Загрузить первый аргумент
gen.Emit (OpCodes.Call, typeof(Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Ret);

Type realType = tb.CreateType();
double x = (double) tb.GetMethod ("SquareRoot").Invoke (null,
                                                  new object[] { 10.0 });
Console.WriteLine (x);                // 3.16227766016838
```

Вызов метода DefineParameter является необязательным и обычно делается для назначения параметру имени. Число 1 ссылается на первый параметр (0 соответствует возвращаемому значению). Если DefineParameter не вызывается, тогда параметры неявно именуются как __p1, __p2 и т.д. Назначение имен имеет смысл, если сборка будет записываться на диск; оно делает методы дружественными к потребителям.

Метод DefineParameter возвращает объект ParameterBuilder, на котором можно вызывать метод SetCustomAttribute для присоединения атрибутов (см. раздел "Присоединение атрибутов" далее в главе).

Для выпуска параметров, передаваемых по ссылке, таких как параметр в следующем методе C#:

```
public static void SquareRoot (ref double value)
{
  value = Math.Sqrt (value);
}
```

необходимо вызвать метод MakeByRefType на типе параметра (или типах):

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  null,
  new Type[] { typeof (double).MakeByRefType() } );

mb.DefineParameter (1, ParameterAttributes.None, "value");

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);
```

```
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ldind_R8);
gen.Emit (OpCodes.Call, typeof (Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Stind_R8);
gen.Emit (OpCodes.Ret);

Type realType = tb.CreateType();
object[] args = { 10.0 };
tb.GetMethod ("SquareRoot").Invoke (null, args);
Console.WriteLine (args[0]);  // 3.16227766016838
```

Здесь коды операций были скопированы из дизассемблированного метода C#. Обратите внимание на разницу в семантике для доступа к параметрам, передаваемым по ссылке: коды операций `Ldind` и `Stind` означают соответственно "загрузить косвенно" (load indirectly) и "сохранить косвенно" (store indirectly). Суффикс `R8` означает 8-байтовое число с плавающей точкой.

Процесс выпуска параметров `out` идентичен за исключением того, что метод `DefineParameter` вызывается следующим образом:

```
mb.DefineParameter (1, ParameterAttributes.Out, "value");
```

### Генерация методов экземпляра

Чтобы сгенерировать метод экземпляра, при вызове `DefineMethod` понадобится указать флаг `MethodAttributes.Instance`:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Instance | MethodAttributes.Public
  ...
```

В случае методов экземпляра нулевым аргументом неявно является `this`; нумерация остальных аргументов начинается с 1. Таким образом, `Ldarg_0` загружает в стек оценки `this`, а `Ldarg_1` загружает первый реальный аргумент метода.

### Переопределение методов

Переопределять виртуальный метод в базовом классе легко: нужно просто определить метод с идентичным именем, сигнатурой и возвращаемым типом, указав при вызове `DefineMethod` флаг `MethodAttributes.Virtual`. То же самое применимо при реализации методов интерфейса.

В классе `TypeBuilder` также открыт метод по имени `DefineMethodOverride`, который переопределяет метод с другим именем. Использовать его имеет смысл только с явной реализацией интерфейса; в остальных сценариях следует применять метод `DefineMethod`.

### Флаг `HideBySig`

В случае построения подкласса другого типа при определении методов почти всегда полезно указывать флаг `MethodAttributes.HideBySig`. Флаг `HideBySig` обеспечивает использование семантики сокрытия методов в стиле C#, которая заключается в том, что метод базового класса скрывается только в случае, если в подтипе определен метод с такой же *сигнатурой*. Без `HideBySig` сокрытие методов основывается только на *имени*, поэтому метод `Foo(string)` в подтипе скроет метод `Foo()` в базовом типе, хотя подобное обычно нежелательно.

## Выпуск полей и свойств

Для создания поля необходимо вызвать метод `DefineField` на объекте `TypeBuilder`, сообщив ему желаемое имя поля, тип и видимость. Следующий код создает закрытое целочисленное поле по имени `length`:

```
FieldBuilder field = tb.DefineField ("length", typeof (int),
                                     FieldAttributes.Private);
```

Создание свойства или индексатора требует выполнения нескольких дополнительных шагов. Первый из них — вызов метода `DefineProperty` на объекте `TypeBuilder` с передачей ему имени и типа свойства:

```
PropertyBuilder prop = tb.DefineProperty (
                       "Text",                          // Имя свойства
                       PropertyAttributes.None,
                       typeof (string),                 // Тип свойства
                       new Type[0]                      // Типы индексатора
                      );
```

(При создании индексатора последний аргумент представляет собой массив типов индексатора.) Обратите внимание, что мы не указываем видимость свойства: это делается в индивидуальном порядке на основе методов аксессора.

Следующий шаг заключается в написании методов `get` и `set`. По соглашению их имена имеют префикс `get_` или `set_`. Затем готовые методы можно присоединить к свойству с помощью вызова методов `SetGetMethod` и `SetSetMethod` на объекте `PropertyBuilder`.

В качестве полного примера мы возьмем показанное ниже объявление поля и свойства:

```
string _text;
public string Text
{
  get          { return _text; }
  internal set { _text = value; }
}
```

и сгенерируем его динамически:

```
FieldBuilder field = tb.DefineField ("_text", typeof (string),
                                     FieldAttributes.Private);
PropertyBuilder prop = tb.DefineProperty (
                       "Text",                          // Имя свойства
                       PropertyAttributes.None,
                       typeof (string),                 // Тип свойства
                       new Type[0]);                    // Типы индексатора

MethodBuilder getter = tb.DefineMethod (
  "get_Text",                                           // Имя метода
  MethodAttributes.Public | MethodAttributes.SpecialName,
  typeof (string),                                      // Возвращаемый тип
  new Type[0]);                                         // Типы параметров

ILGenerator getGen = getter.GetILGenerator();
getGen.Emit (OpCodes.Ldarg_0);          // Загрузить в стек оценки this
getGen.Emit (OpCodes.Ldfld, field);     // Загрузить в стек оценки значение
свойства
getGen.Emit (OpCodes.Ret);              // Выполнить возвращение
```

```
MethodBuilder setter = tb.DefineMethod (
  "set_Text",
  MethodAttributes.Assembly | MethodAttributes.SpecialName,
  null,                                   // Возвращаемый тип
  new Type[] { typeof (string) } );       // Типы параметров

ILGenerator setGen = setter.GetILGenerator();
setGen.Emit (OpCodes.Ldarg_0);  // Загрузить в стек оценки this
setGen.Emit (OpCodes.Ldarg_1);  // Загрузить в стек оценки второй аргумент,
                                // т.е. значение
setGen.Emit (OpCodes.Stfld, field);     // Сохранить значение в поле
setGen.Emit (OpCodes.Ret);              // Выполнить возвращение

prop.SetGetMethod (getter);             // Связать метод get и свойство
prop.SetSetMethod (setter);             // Связать метод set и свойство
```

Теперь свойство можно протестировать:

```
Type t = tb.CreateType();
object o = Activator.CreateInstance (t);
t.GetProperty ("Text").SetValue (o, "Good emissions!", new object[0]);
string text = (string) t.GetProperty ("Text").GetValue (o, null);

Console.WriteLine (text);  // Good emissions!
```

Обратите внимание, что в определении `MethodAttributes` для аксессора был включен флаг `SpecialName`. Он инструктирует компилятор о том, что прямое связывание с такими методами при статической ссылке на сборку не разрешено. Это также гарантирует соответствующую поддержку аксессоров инструментами рефлексии и средством IntelliSense в Visual Studio.

Выпускать события можно аналогично, вызывая метод `DefineEvent` на объекте `TypeBuilder`. Затем можно написать явные методы аксессора и присоединить их к объекту `EventBuilder` путем вызова методов `SetAddOnMethod` и `SetRemoveOnMethod`.

## Выпуск конструкторов

Чтобы определить собственные конструкторы, понадобится вызвать метод `DefineConstructor` на объекте `TypeBuilder`. Поступать так не обязательно – стандартный конструктор без параметров будет предоставлен автоматически, если не было явно определено ни одного конструктора. В случае подтипа стандартный конструктор вызывает конструктор базового класса – точно как в C#. Определение одного или большего числа конструкторов приводит к устранению стандартного конструктора.

Конструктор является удобным местом для инициализации полей. На самом деле он представляет собой единственное такое место: инициализаторы полей C# не имеют специальной поддержки в CLR – это просто синтаксическое сокращение для присваивания значений полям в конструкторе.

Таким образом, для воспроизведения следующего кода:

```
class Widget
{
  int _capacity = 4000;
}
```

потребуется определить конструктор, как показано ниже:

```
FieldBuilder field = tb.DefineField ("_capacity", typeof (int),
                                     FieldAttributes.Private);
ConstructorBuilder c = tb.DefineConstructor (
  MethodAttributes.Public,
  CallingConventions.Standard,
  new Type[0]);                      // Параметры конструктора

ILGenerator gen = c.GetILGenerator();

gen.Emit (OpCodes.Ldarg_0);          // Загрузить в стек оценки this
gen.Emit (OpCodes.Ldc_I4, 4000);     // Загрузить в стек оценки 4000
gen.Emit (OpCodes.Stfld, field);     // Сохранить это в поле field
gen.Emit (OpCodes.Ret);
```

### Вызов конструкторов базовых классов

При построении подкласса другого типа конструктор, который был только что написан, *обойдет конструктор базового класса*. Ситуация отличается от C#, где конструктор базового класса вызывается всегда, прямо или косвенно. Например, имея приведенный далее код:

```
class A     { public A() { Console.Write ("A"); } }
class B : A { public B() {} }
```

компилятор в действительности будет транслировать вторую строку следующим образом:

```
class B : A { public B() : base() {} }
```

Однако это не так, когда генерируется код IL: если нужно, чтобы конструктор базового класса был выполнен, то он должен вызываться явно (что происходит почти всегда). Предполагая, что базовый класс имеет имя A, вот как нужно сделать:

```
gen.Emit (OpCodes.Ldarg_0);
ConstructorInfo baseConstr = typeof (A).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Call, baseConstr);
```

Конструкторы с аргументами вызываются точно так же, как обычные методы.

## Присоединение атрибутов

Присоединить специальные атрибуты к динамической конструкции можно путем вызова метода `SetCustomAttribute` с передачей ему объекта `CustomAttributeBuilder`. Например, пусть необходимо присоединить к полю или свойству следующее объявление атрибута:

```
[XmlElement ("FirstName", Namespace="http://test/", Order=3)]
```

Объявление полагается на конструктор класса `XmlElementAttribute`, который принимает одиночную строку. Для работы с объектом `CustomAttributeBuilder` потребуется извлечь как указанный конструктор, так и два дополнительных свойства, подлежащие установке (`Namespace` и `Order`):

```
Type attType = typeof (XmlElementAttribute);

ConstructorInfo attConstructor = attType.GetConstructor (
  new Type[] { typeof (string) } );

var att = new CustomAttributeBuilder (
  attConstructor,                    // Конструктор
  new object[] { "FirstName" },      // Аргументы конструктора
```

```
    new PropertyInfo[]
    {
      attType.GetProperty ("Namespace"),  // Свойства
      attType.GetProperty ("Order")
    },
    new object[] { "http://test/", 3 }    // Значения свойств
  );
  myFieldBuilder.SetCustomAttribute (att);
  // или propBuilder.SetCustomAttribute (att);
  // или typeBuilder.SetCustomAttribute (att);  и т.д.
```

# Выпуск обобщенных методов и типов

Во всех примерах раздела предполагается, что объект `modBuilder` был создан следующим образом:

```
AppDomain domain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName ("MyEmissions");

AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
  "MainModule", "MyEmissions.dll");
```

## Определение обобщенных методов

Для выпуска обобщенного метода выполните перечисленные шаги.

1. Вызовите метод `DefineGenericParameters` на объекте `MethodBuilder`, чтобы получить массив объектов `GenericTypeParameterBuilder`.
2. Вызовите метод `SetSignature` на объекте `MethodBuilder` с применением этих параметров обобщенных типов (т.е. массива объектов `GenericTypeParameterBuilder`).
3. При желании назначьте параметрам другие имена.

Например, следующий обобщенный метод:

```
public static T Echo<T> (T value)
{
  return value;
}
```

можно было бы сгенерировать так:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);

MethodBuilder mb = tb.DefineMethod ("Echo", MethodAttributes.Public |
                                    MethodAttributes.Static);
GenericTypeParameterBuilder[] genericParams
  = mb.DefineGenericParameters ("T");

mb.SetSignature (genericParams[0],     // Возвращаемый тип
                 null, null,
                 genericParams,        // Типы параметров
                 null, null);

mb.DefineParameter (1, ParameterAttributes.None, "value"); // Необязательно
```

```
ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ret);
```

Метод DefineGenericParameters принимает любое количество строковых аргументов — они соответствуют именам желаемых обобщенных типов. В данном примере необходим только один обобщенный тип по имени T. Класс GenericTypeParameterBuilder основан на System.Type, поэтому он может использоваться на месте TypeBuilder при выпуске кодов операций.

Класс GenericTypeParameterBuilder также позволяет указывать ограничение базового типа:

```
genericParams[0].SetBaseTypeConstraint (typeof (Foo));
```

и ограничения интерфейсов:

```
genericParams[0].SetInterfaceConstraints (typeof (IComparable));
```

Чтобы воспроизвести приведенный ниже код:

```
public static T Echo<T> (T value) where T : IComparable<T>
```

потребуется записать так:

```
genericParams[0].SetInterfaceConstraints (
  typeof (IComparable<>).MakeGenericType (genericParams[0]) );
```

Для других видов ограничений нужно вызывать метод SetGenericParameterAttributes. Он принимает член перечисления GenericParameterAttributes, которое содержит следующие значения:

- DefaultConstructorConstraint
- NotNullableValueTypeConstraint
- ReferenceTypeConstraint
- Covariant
- Contravariant

Последние два значения эквивалентны применению к параметрам типа модификаторов out и in.

## Определение обобщенных типов

Обобщенные типы определяются в похожей манере. Отличие заключается в том, что метод DefineGenericParameters вызывается на объекте TypeBuilder, а не MethodBuilder. Таким образом, для воспроизведения следующего определения:

```
public class Widget<T>
{
  public T Value;
}
```

потребуется написать такой код:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
GenericTypeParameterBuilder[] genericParams
  = tb.DefineGenericParameters ("T");
tb.DefineField ("Value", genericParams[0], FieldAttributes.Public);
```

Как и в случае методов, можно добавлять обобщенные ограничения.

# Сложности, связанные с генерацией

Во всех примерах данного раздела предполагается, что объект `modBuilder` создавался, как было показано в предшествующих разделах.

## Несозданные закрытые обобщения

Пусть необходимо сгенерировать метод, который использует закрытый обобщенный тип:

```
public class Widget
{
  public static void Test() { var list = new List<int>(); }
}
```

Процесс довольно прямолинеен:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
                                    MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();
Type variableType = typeof (List<int>);
ConstructorInfo ci = variableType.GetConstructor (new Type[0]);
LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

Теперь предположим, что вместо списка целых чисел требуется список объектов `Widget`:

```
public class Widget
{
  public static void Test() { var list = new List<Widget>(); }
}
```

Теоретически модификация проста — нужно лишь заменить строку:

```
Type variableType = typeof (List<int>);
```

строкой:

```
Type variableType = typeof (List<>).MakeGenericType (tb);
```

К сожалению, при последующем вызове метода `GetConstructor` сгенерируется исключение `NotSupportedException`. Проблема в том, что вызывать `GetConstructor` на обобщенном типе, закрытом с помощью несозданного построителя типа, не допускается. То же самое касается методов `GetField` и `GetMethod`.

Решение нельзя считать интуитивно понятным. В классе `TypeBuilder` присутствуют следующие три статических метода:

```
public static ConstructorInfo GetConstructor (Type, ConstructorInfo);
public static FieldInfo      GetField      (Type, FieldInfo);
public static MethodInfo     GetMethod     (Type, MethodInfo);
```

Хотя методы таковыми не выглядят, они существуют специально для получения членов обобщенных типов, закрытых посредством несозданных построителей типов!

Первый параметр представляет собой закрытый обобщенный тип, а второй параметр — желаемый член из *несвязанного* обобщенного типа. Ниже приведена скорректированная версия рассматриваемого примера:

```
MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
                                    MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();
Type variableType = typeof (List<>).MakeGenericType (tb);
ConstructorInfo unbound = typeof (List<>).GetConstructor (new Type[0]);
ConstructorInfo ci = TypeBuilder.GetConstructor (variableType, unbound);
LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

## Циклические зависимости

Предположим, что нужно построить два типа, которые ссылаются друг на друга. Например:

```
class A { public B Bee; }
class B { public A Aye; }
```

Сгенерировать это динамически можно так:

```
var publicAtt = FieldAttributes.Public;

TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");
FieldBuilder bee = aBuilder.DefineField ("Bee", bBuilder, publicAtt);
FieldBuilder aye = bBuilder.DefineField ("Aye", aBuilder, publicAtt);
Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();
```

Обратите внимание, что мы не вызывали метод `CreateType` на объекте `aBuilder` или `bBuilder`, пока оба объекта не были заполнены. Здесь применяется следующий принцип: сначала все связывается, а затем производится вызов метода `CreateType` на каждом построителе типа.

Интересно отметить, что тип `realA` является допустимым, но *дисфункциональным* до тех пор, пока не будет вызван метод `CreateType` на `bBuilder`. (Если вы начнете использовать объект `aBuilder` до такого момента, то при попытке доступа к полю `Bee` сгенерируется исключение.)

Вас может заинтересовать, каким образом `bBuilder` узнает о необходимости "исправления" типа `realA` после создания `realB`? На самом деле он вовсе не знает об этом: тип `realA` может исправить себя *самостоятельно* при следующем его применении. Исправление возможно из-за того, что после вызова метода `CreateType` объект `TypeBuilder` превращается в прокси для действительного типа времени выполнения. Таким образом, благодаря своим ссылкам на `bBuilder` тип `realA` может легко получить метаданные, требующиеся для обновления.

Описанная система работает, когда построитель типа запрашивает простую информацию о несозданном типе — информацию, которая может быть *предварительно определена* — такую как тип, член и объектные ссылки. При создании `realA` построителю типа не нужно знать, скажем, сколько байтов памяти будет в итоге занимать `realB`. И это вполне нормально, т.к. тип `realB` пока еще не создан! Но теперь представьте, что тип `realB` был структурой. Окончательный размер `realB` теперь становится критически важной информацией при создании типа `realA`.

Если отношение между типами не является циклическим, например:

```
struct A { public B Bee; }
struct B {               }
```

то задачу можно решить, сначала создав структуру B, а затем структуру A. Но взгляните на следующие определения:

```
struct A { public B Bee; }
struct B { public A Aye; }
```

Мы даже не будем пытаться выпустить такой код, поскольку определение двух структур, содержащих друг друга, лишено смысла (компилятор C# сгенерирует ошибку на этапе компиляции). Но показанная далее вариация как законна, так и полезна:

```
public struct S<T> { ... }     // Структура S может быть пустой
                               // и эта демонстрация будет работать.

class A { S<B> Bee; }
class B { S<A> Aye; }
```

При создании класса A построитель типа теперь должен располагать знанием отпечатка памяти класса B и наоборот. В целях иллюстрации предположим, что структура S определена статически. Код для выпуска классов A и B мог бы выглядеть так:

```
var pub = FieldAttributes.Public;

TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");

aBuilder.DefineField ("Bee", typeof(S<>).MakeGenericType (bBuilder), pub);
bBuilder.DefineField ("Aye", typeof(S<>).MakeGenericType (aBuilder), pub);

Type realA = aBuilder.CreateType();  // Ошибка: не удается загрузить тип B
Type realB = bBuilder.CreateType();
```

Независимо от порядка выполнения метод `CreateType` генерирует исключение `TypeLoadException`:

- если первым идет вызов `aBuilder.CreateType`, то исключение сообщает о невозможности загрузки типа B;
- если первым идет вызов `bBuilder.CreateType`, то исключение сообщает о невозможности загрузки типа A.

Вы столкнетесь с данной проблемой при динамическом выпуске типизированных объектов `DataContext` для LINQ to SQL. Обобщенный тип `EntityRef` является структурой, которая эквивалентна структуре S в приведенных ранее примерах. Циклическая ссылка возникает, когда две таблицы в базе данных ссылаются друг на друга через взаимные отношения "родительский/дочерний".

Чтобы решить проблему, вы должны позволить построителю типа создать `realB` частично через создание `realA`. Это делается за счет обработки события `TypeResolve` в текущем домене приложения непосредственно перед вызовом метода `CreateType`. Таким образом, в рассматриваемом примере мы заменяем последние две строки следующим кодом:

```
TypeBuilder[] uncreatedTypes = { aBuilder, bBuilder };

ResolveEventHandler handler = delegate (object o, ResolveEventArgs args)
{
  var type = uncreatedTypes.FirstOrDefault (t => t.FullName == args.Name);
  return type == null ? null : type.CreateType().Assembly;
};
```

```
AppDomain.CurrentDomain.TypeResolve += handler;

Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();

AppDomain.CurrentDomain.TypeResolve -= handler;
```

Событие TypeResolve инициируется во время вызова метода aBuilder.CreateType, в точке, где нужно, чтобы вы вызвали CreateType на bBuilder.

Обработка события TypeResolve, как в представленном примере, также необходима при определении вложенного типа, когда вложенный и родительский типы ссылаются друг на друга.

# Синтаксический разбор IL

Для получения информации о содержимом существующего метода понадобится вызвать метод GetMethodBody на объекте MethodBase. Вызов возвращает объект MethodBody, который имеет свойства для инспектирования локальных переменных метода, конструкций обработки исключений, размера стека, а также низкоуровневого кода IL. Очень похоже на противоположность метода Reflection.Emit!

Инспектирование низкоуровневого кода IL метода может быть полезно при профилировании кода. Простой сценарий использования мог бы предусматривать выяснение, какие методы в сборке изменились в результате ее обновления.

Для демонстрации синтаксического разбора IL мы напишем приложение, которое дизассемблирует код IL, работая в стиле ildasm. Приложение подобного рода могло бы служить отправной точкой для построения инструмента анализа кода или дизассемблера языка более высокого уровня.

Вспомните, что в API-интерфейсе рефлексии все функциональные конструкции C# либо представлены подтипом MethodBase, либо (в случае свойств, событий и индексаторов) имеют присоединенные к ним объекты MethodBase.

## Написание дизассемблера

Исходный код доступен для загрузки по адресу http://www.albahari.com/nutshell/.

Ниже приведен пример вывода, который будет производить наш дизассемблер:

```
IL_00EB:  ldfld      Disassembler._pos
IL_00F0:  ldloc.2
IL_00F1:  add
IL_00F2:  ldelema     System.Byte
IL_00F7:  ldstr      "Hello world"
IL_00FC:  call       System.Byte.ToString
IL_0101:  ldstr      " "
IL_0106:  call       System.String.Concat
```

Чтобы получить такой вывод, потребуется провести синтаксический разбор двоичных лексем, из которых сформирован код IL. Первый шаг заключается в вызове метода GetILAsByteArray на объекте MethodBody для получения кода IL в виде бай-

тового массива. Для упрощения оставшейся работы мы реализуем решение такой задачи в форме класса:

```
public class Disassembler
{
  public static string Disassemble (MethodBase method)
    => new Disassembler (method).Dis();

  StringBuilder _output;  // Результат, который будет постоянно дополняться
  Module _module;         // Это пригодится в дальнейшем
  byte[] _il;             // Низкоуровневый байтовый код
  int _pos;               // Позиция внутри байтового кода
  Disassembler (MethodBase method)
  {
    _module = method.DeclaringType.Module;
    _il = method.GetMethodBody().GetILAsByteArray();
  }
  string Dis()
  {
   _output = new StringBuilder();
    while (_pos < _il.Length) DisassembleNextInstruction();
    return _output.ToString();
  }
}
```

Статический метод `Disassemble` будет единственным открытым членом в данном классе. Все другие члены будут закрытыми по отношению к процессу дизассемблирования. Метод `Dis` содержит "главный" цикл, в котором мы обрабатываем каждую инструкцию.

Имея такой скелет, остается лишь написать метод `DisassembleNextInstruction`. Но перед тем как делать это, полезно загрузить все коды операций в статический словарь, чтобы к ним можно было обращаться по их 8- или 16-битным значениям. Простейший способ достичь такой цели — воспользоваться рефлексией для извлечения всех статических полей типа `OpCode` из класса `OpCodes`:

```
static Dictionary<short,OpCode> _opcodes = new Dictionary<short,OpCode>();

static Disassembler()
{
  Dictionary<short, OpCode> opcodes = new Dictionary<short, OpCode>();
    foreach (FieldInfo fi in typeof (OpCodes).GetFields
                          (BindingFlags.Public | BindingFlags.Static))
      if (typeof (OpCode).IsAssignableFrom (fi.FieldType))
      {
        OpCode code = (OpCode) fi.GetValue (null);  // Получить значение поля
        if (code.OpCodeType != OpCodeType.Nternal)
          _opcodes.Add (code.Value, code);
      }
}
```

Мы поместили код в статический конструктор, так что он будет выполняться только один раз.

Теперь можно заняться реализацией метода `DisassembleNextInstruction`. Каждая инструкция IL состоит из однобайтового или двухбайтового кода операции, за которым следует операнд длиной 0, 1, 2, 4 или 8 байтов. (Исключением являются коды операций встроенных переключателей, за которыми следует переменное количество операндов.)

Итак, мы читаем код операции, далее операнд и затем выводим результат:

```
void DisassembleNextInstruction()
{
  int opStart = _pos;
  OpCode code = ReadOpCode();
  string operand = ReadOperand (code);
  _output.AppendFormat ("IL_{0:X4}:  {1,-12} {2}", opStart, code.Name, operand);
  _output.AppendLine();
}
```

Для чтения кода операции мы продвигаемся вперед на один байт и выясняем, является ли он допустимой инструкцией. Если нет, тогда мы продвигаемся вперед еще на один байт и проверяем, существует ли двухбайтовая инструкция:

```
OpCode ReadOpCode()
{
  byte byteCode = _il [_pos++];
  if (_opcodes.ContainsKey (byteCode)) return _opcodes [byteCode];
  if (_pos == _il.Length)  throw new Exception ("Unexpected end of IL");
                                                  // Неожиданный конец кода IL
  short shortCode = (short) (byteCode * 256 + _il [_pos++]);
  if (!_opcodes.ContainsKey (shortCode))
    throw new Exception ("Cannot find opcode " + shortCode);
  return _opcodes [shortCode];
}
```

Чтобы прочитать операнд, сначала потребуется выяснить его длину. Это можно сделать на основе типа операнда. Поскольку большинство их них имеют 4 байта в длину, отклонения можно довольно легко отфильтровать в условной конструкции.

Следующий шаг заключается в вызове метода `FormatOperand`, который попытается сформатировать операнд:

```
string ReadOperand (OpCode c)
{
  int operandLength =
    c.OperandType == OperandType.InlineNone
      ? 0 :
    c.OperandType == OperandType.ShortInlineBrTarget ||
    c.OperandType == OperandType.ShortInlineI ||
    c.OperandType == OperandType.ShortInlineVar
      ? 1 :
    c.OperandType == OperandType.InlineVar
      ? 2 :
    c.OperandType == OperandType.InlineI8 ||
    c.OperandType == OperandType.InlineR
      ? 8 :
    c.OperandType == OperandType.InlineSwitch
      ? 4 * (BitConverter.ToInt32 (_il, _pos) + 1) :
      4;  // Все остальные имеют длину 4 байта
  if (_pos + operandLength > _il.Length)
    throw new Exception ("Unexpected end of IL");
                        // Неожиданный конец кода IL
  string result = FormatOperand (c, operandLength);
  if (result == null)
  {  // Вывести байты операнда в шестнадцатеричном виде
    result = "";
```

```
    for (int i = 0; i < operandLength; i++)
      result += _il [_pos + i].ToString ("X2") + " ";
  }
  _pos += operandLength;
  return result;
}
```

Если после вызова метода FormatOperand значение result равно null, то это означает, что операнд не нуждается в специальном форматировании, и мы просто выводим его в шестнадцатеричном виде. Мы могли бы протестировать дизассемблер в данной точке, написав метод FormatOperand, который всегда возвращает null. Ниже показано, как будет выглядеть вывод:

```
IL_00A8:  ldfld      98 00 00 04
IL_00AD:  ldloc.2
IL_00AE:  add
IL_00AF:  ldelema    64 00 00 01
IL_00B4:  ldstr      26 04 00 70
IL_00B9:  call       B6 00 00 0A
IL_00BE:  ldstr      11 01 00 70
IL_00C3:  call       91 00 00 0A
...
```

Хотя коды операций корректны, операнды в таком виде не особенно полезны. Вместо шестнадцатеричных цифр нам необходимы имена членов и строки. После реализации метод FormatOperand решит проблему, идентифицируя специальные случаи, которые выигрывают от такого форматирования. Они включают большинство 4-байтовых операндов и сокращенные инструкции ветвления:

```
string FormatOperand (OpCode c, int operandLength)
{
  if (operandLength == 0) return "";
  if (operandLength == 4)
    return Get4ByteOperand (c);
  else if (c.OperandType == OperandType.ShortInlineBrTarget)
    return GetShortRelativeTarget();
  else if (c.OperandType == OperandType.InlineSwitch)
    return GetSwitchTarget (operandLength);
  else
    return null;
}
```

Есть три вида 4-байтовых операндов, которые мы трактуем специальным образом. Первый вид относится к членам или типам — в данном случае мы извлекаем имя члена или типа, вызывая метод ResolveMember определяющего модуля. Второй вид — строки; они хранятся в метаданных модуля сборки и могут быть извлечены вызовом метода ResolveString. Третий вид касается целей ветвления, когда операнды ссылаются на байтовое смещение в коде IL. Мы форматируем их за счет работы с абсолютным адресом *после* текущей инструкции (+ 4 байта):

```
string Get4ByteOperand (OpCode c)
{
  int intOp = BitConverter.ToInt32 (_il, _pos);
  switch (c.OperandType)
  {
    case OperandType.InlineTok:
    case OperandType.InlineMethod:
    case OperandType.InlineField:
```

```
    case OperandType.InlineType:
      MemberInfo mi;
      try   { mi = _module.ResolveMember (intOp); }
      catch { return null; }
      if (mi == null) return null;
      if (mi.ReflectedType != null)
        return mi.ReflectedType.FullName + "." + mi.Name;
      else if (mi is Type)
        return ((Type)mi).FullName;
      else
        return mi.Name;
    case OperandType.InlineString:
      string s = _module.ResolveString (intOp);
      if (s != null) s = "'" + s + "'";
      return s;
    case OperandType.InlineBrTarget:
      return "IL_" + (_pos + intOp + 4).ToString ("X4");
    default:
      return null;
  }
}
```

Точка, где мы вызываем `ResolveMember`, представляет собой хорошее окно для инструмента анализа кода, который сообщает о зависимостях методов.

Для любого другого 4-байтового кода операции мы возвращаем `null` (что заставляет метод `ReadOperand` форматировать операнд в виде шестнадцатеричных цифр).

Последняя разновидность операндов, которая требует особого внимания – сокращенные цели ветвления и встроенные переключатели. Сокращенная цель ветвления описывает смещение назначения в виде одиночного байта со знаком, как в конце текущей инструкции (т.е. + 1 байт). За целью переключателя следует переменное количество 4-байтовых целей ветвления:

```
string GetShortRelativeTarget()
{
  int absoluteTarget = _pos + (sbyte) _il [_pos] + 1;
  return "IL_" + absoluteTarget.ToString ("X4");
}
string GetSwitchTarget (int operandLength)
{
  int targetCount = BitConverter.ToInt32 (_il, _pos);
  string [] targets = new string [targetCount];
  for (int i = 0; i < targetCount; i++)
  {
    int ilTarget = BitConverter.ToInt32 (_il, _pos + (i + 1) * 4);
    targets [i] = "IL_" + (_pos + ilTarget + operandLength).ToString ("X4");
  }
  return "(" + string.Join (", ", targets) + ")";
}
```

На этом дизассемблер завершен. Чтобы протестировать класс `Disassembler`, можно дизассемблировать один из его собственных методов:

```
MethodInfo mi = typeof (Disassembler).GetMethod (
  "ReadOperand", BindingFlags.Instance | BindingFlags.NonPublic);
Console.WriteLine (Disassembler.Disassemble (mi));
```

# 20

# Динамическое программирование

В главе 4 мы объяснили, как работает динамическое связывание в языке C#. В настоящей главе мы кратко рассмотрим среду DLR, после чего раскроем следующие паттерны динамического программирования:

- унификация числовых типов;
- динамическое распознавание перегруженных членов;
- специальное связывание (реализация динамических объектов);
- взаимодействие с динамическими языками.

В главе 25 мы покажем, каким образом ключевое слово `dynamic` может улучшить взаимодействие с COM.

Типы, рассматриваемые в главе, находятся в пространстве имен `System.Dynamic` за исключением типа `CallSite<>`, который определен в пространстве имен `System.Runtime.CompilerServices`.

## Исполняющая среда динамического языка

При выполнении динамического связывания язык C# полагается на *исполняющую среду динамического языка* (Dynamic Language Runtime – DLR).

Несмотря на свое название, DLR не является динамической версией среды CLR. В действительности она представляет собой библиотеку, которая функционирует поверх CLR – точно так же, как любая другая библиотека вроде `System.Xml.dll`. Ее основная роль – предоставлять службы времени выполнения для *унификации* динамического программирования на статически и динамически типизированных языках. Следовательно, такие языки, как C#, VB, Iron-Python и IronRuby, используют один и тот же протокол для вызова функций динамическим образом. Это позволяет им разделять библиотеки и обращаться к коду, написанному на других языках.

Среда DLR также позволяет сравнительно легко создавать новые динамические языки в .NET. Вместо выпуска кода IL авторы динамических языков работают на

уровне *деревьев выражений* (тех же самых деревьев выражений из пространства имен `System.Linq.Expressions`, которые обсуждались в главе 8).

Среда DLR дополнительно гарантирует, что все потребители получают преимущество *кеширования места вызова*, представляющего собой оптимизацию, в соответствии с которой DLR избегает излишнего повторения потенциально затратных действий по распознаванию членов, предпринимаемых во время динамического связывания.

.NET Framework 4.0 была первой версией инфраструктуры .NET Framework, в состав которой вошла среда DLR. Ранее DLR существовала как отдельная загрузка на сайте Codeplex. Данный сайт по-прежнему содержит ряд дополнительных полезных ресурсов для разработчиков языков.

## Что такое место вызова?

Когда компилятор встречает динамическое выражение, он не имеет никакого представления о том, кто или что будет оценивать это выражение во время выполнения. Например, рассмотрим следующий метод:

```
public dynamic Foo (dynamic x, dynamic y)
{
  return x / y;  // Динамическое выражение
}
```

Переменные x и y могут быть любыми объектами CLR, объектами COM или даже объектами, размещенными в среде какого-то динамического языка. Таким образом, компилятор не в состоянии применить обычный статический подход с выпуском вызова известного метода из известного типа. Взамен компилятор выпускает код, который в итоге дает дерево выражения. Такое дерево выражения описывает операцию, управляемую *местом вызова* (call site), к которому среда DLR привяжется во время выполнения. По существу место вызова действует как посредник между вызывающим и вызываемым компонентами.

Место вызова представлено классом `CallSite<>` из сборки `System.Core.dll`. В этом можно убедиться, дизассемблировав предыдущий метод; результат будет выглядеть приблизительно так:

```
static CallSite<Func<CallSite,object,object,object>> divideSite;

[return: Dynamic]
public object Foo ([Dynamic] object x, [Dynamic] object y)
{
  if (divideSite == null)
    divideSite =
      CallSite<Func<CallSite,object,object,object>>.Create (
        Microsoft.CSharp.RuntimeBinder.Binder.BinaryOperation (
          CSharpBinderFlags.None,
          ExpressionType.Divide,
          /* Для краткости остальные аргументы не показаны */ ));

  return divideSite.Target (divideSite, x, y);
}
```

Как видите, место вызова кешируется в статическом поле, чтобы избежать накладных расходов, обусловленных его повторным созданием в каждом вызове. Среда DLR дополнительно кеширует результат фазы привязки и фактические целевые объекты метода. (В зависимости от типов x и y может существовать множество целевых объектов.)

Затем происходит действительный динамический вызов за счет обращения к полю `Target` (делегат) места вызова с передачей ему операндов `x` и `y`.

Обратите внимание, что класс `Binder` специфичен для C#. Каждый язык с поддержкой динамического связывания предоставляет специфичный для языка связыватель, помогающий среде DLR интерпретировать выражения в манере, которая присуща языку и не является неожиданной для программиста. Например, если мы вызываем метод `Foo` с целочисленными значениями `5` и `2`, то связыватель C# обеспечит получение обратно значения `2`. В противоположность этому связыватель VB.NET приведет к возвращению значения `2.5`.

# Унификация числовых типов

В главе 4 было показано, что ключевое слово `dynamic` позволяет написать единственный метод, который работает со всеми числовыми типами:

```
static dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;

static void Main()
{
  int x = 3, y = 5;
  Console.WriteLine (Mean (x, y));
}
```

Довольно забавен тот факт, что в C# ключевые слова `static` и `dynamic` могут появляться рядом друг с другом! То же самое касается ключевых слов `internal` и `extern`.

Тем не менее, в итоге (неизбежно) приносится в жертву безопасность типов. Следующий код скомпилируется без ошибок, но потерпит неудачу во время выполнения:

```
string s = Mean (3, 5);  // Ошибка во время выполнения!
```

Ситуацию можно исправить, для чего ввести параметр обобщенного типа и затем привести его к `dynamic` внутри самого вычисления:

```
static T Mean<T> (T x, T y)
{
  dynamic result = ((dynamic) x + y) / 2;
  return (T) result;
}
```

Важно отметить, что мы *явно* приводим результат обратно к типу `T`. Если опустить такое приведение, тогда мы будем полагаться на неявное приведение, которое на первый взгляд может показаться работающим корректно. Однако во время выполнения неявное приведение откажет при вызове метода с 8- или 16-битным целочисленным типом. Чтобы понять причину, посмотрим, что происходит с обычной статической типизацией, когда производится суммирование двух 8-битных чисел:

```
byte b = 3;
Console.WriteLine ((b + b).GetType().Name);  // Int32
```

Мы получаем результат типа `Int32`, т.к. перед выполнением арифметических операций компилятор "повышает" 8- или 16-битные числа до `Int32`. Для обеспечения согласованности связыватель C# сообщает среде DLR о необходимости поступать точно так же, и

мы в итоге получаем значение Int32, которое требует явного приведения к меньшему числовому типу. Разумеется, если мы, скажем, суммируем значения, а не вычисляем их среднее арифметическое, то появляется возможность переполнения.

Динамическое связывание приводит к небольшому снижению производительности – даже с кешированием места вызова. Такого снижения можно избежать, добавив статически типизированные перегруженные версии метода, которые охватывают только самые распространенные типы. Например, если последующее профилирование производительности показывает, что вызов метода Mean со значениями типа double является узким местом, то можно добавить следующую перегруженную версию:

```
static double Mean (double x, double y) => (x + y) / 2;
```

Компилятор отдаст предпочтение этой перегруженной версии, когда метод Mean вызывается с аргументами, для которых на этапе компиляции известно, что они относятся к типу double.

# Динамическое распознавание перегруженных членов

Вызов статически известных методов с динамически типизированными аргументами откладывает распознавание перегруженных членов с этапа компиляции до времени выполнения. Такой подход содействует упрощению решения определенных задач программирования вроде реализации паттерна проектирования "Посетитель" (Visitor). Кроме того, он удобен для обхода ограничений, накладываемых статической типизацией языка C#.

## Упрощение паттерна "Посетитель"

По существу паттерн "Посетитель" позволяет "добавлять" метод в иерархию классов, не изменяя существующие классы. Несмотря на полезность, данный паттерн в своем статическом воплощении является неочевидным и не интуитивно понятным по сравнению с большинством других паттернов проектирования. Он также требует, чтобы посещаемые классы были сделаны "дружественными к паттерну 'Посетитель'" за счет открытия доступа к методу Accept, что может оказаться невозможным, если классы находятся вне вашего контроля.

Посредством динамического связывания той же самой цели можно достигнуть более просто – и без необходимости в модификации существующих классов. В качестве иллюстрации рассмотрим следующую иерархию классов:

```
class Person
{
  public string FirstName { get; set; }
  public string LastName  { get; set; }

  // Коллекция Friends может содержать объекты Customer и Employee:
  public readonly IList<Person> Friends = new Collection<Person> ();
}

class Customer : Person { public decimal CreditLimit { get; set; } }
class Employee : Person { public decimal Salary      { get; set; } }
```

Предположим, что требуется написать метод, который программно экспортирует детали объекта Person в XML-элемент (объект XElement). Наиболее очевидное решение предусматривает реализацию внутри класса Person виртуального метода по

имени ToXElement, который возвращает объект XElement, заполненный значениями свойств объекта Person. Затем метод ToXElement в классах Customer и Employee можно было бы переопределить, чтобы объект XElement также заполнялся значениями свойств CreditLimit и Salary. Однако такой паттерн может оказаться проблематичным по двум причинам.

- Вы можете не владеть кодом классов Person, Customer и Employee, что делает невозможным добавление к ним методов. (А расширяющие методы не обеспечивают полиморфное поведение.)
- Классы Person, Customer и Employee могут уже быть довольно большими. Часто встречающимся антипаттерном является "Божественный объект" ("God Object"), при котором класс, подобный Person, возлагает на себя настолько много функциональности, что его сопровождение превращается в самый настоящий кошмар. Хорошее противодействие этому – избегание добавления в класс Person функций, которым не нужен доступ к закрытому состоянию Person. Великолепным кандидатом может служить метод ToXElement.

Благодаря динамическому распознаванию перегруженных членов мы можем реализовать функциональность метода ToXElement в отдельном классе, не прибегая к неуклюжим операторам switch на основе типа:

```
class ToXElementPersonVisitor
{
  public XElement DynamicVisit (Person p) => Visit ((dynamic)p);

  XElement Visit (Person p)
  {
    return new XElement ("Person",
      new XAttribute ("Type", p.GetType().Name),
      new XElement ("FirstName", p.FirstName),
      new XElement ("LastName", p.LastName),
      p.Friends.Select (f => DynamicVisit (f))
    );
  }

  XElement Visit (Customer c) // Специализированная логика для объектов Customer
  {
    XElement xe = Visit ((Person)c);  // Вызов "базового" метода
    xe.Add (new XElement ("CreditLimit", c.CreditLimit));
    return xe;
  }

  XElement Visit (Employee e) // Специализированная логика для объектов Employee
  {
    XElement xe = Visit ((Person)e);  // Вызов "базового" метода
    xe.Add (new XElement ("Salary", e.Salary));
    return xe;
  }
}
```

Метод DynamicVisit осуществляет динамическую диспетчеризацию – вызывает наиболее специфическую версию метода Visit, как определено во время выполнения. Обратите внимание на выделенную полужирным строку кода, в которой мы вызываем DynamicVisit на каждом объекте Person в коллекции Friends. Такой прием гарантирует, что если элемент коллекции Friends является объектом Customer или Employee, то будет вызвана корректная перегруженная версия метода.

Продемонстрировать использование класса `ToXElementPersonVisitor` можно следующим образом:

```
var cust = new Customer
{
  FirstName = "Joe", LastName = "Bloggs", CreditLimit = 123
};
cust.Friends.Add (
  new Employee { FirstName = "Sue", LastName = "Brown", Salary = 50000 }
);
Console.WriteLine (new ToXElementPersonVisitor().DynamicVisit (cust));
```

Вот как выглядит результат:

```
<Person Type="Customer">
  <FirstName>Joe</FirstName>
  <LastName>Bloggs</LastName>
  <Person Type="Employee">
    <FirstName>Sue</FirstName>
    <LastName>Brown</LastName>
    <Salary>50000</Salary>
  </Person>
  <CreditLimit>123</CreditLimit>
</Person>
```

## Вариации

Если планируется работа с несколькими классами посетителя, тогда удобная вариация предусматривает определение абстрактного базового класса для посетителей:

```
abstract class PersonVisitor<T>
{
  public T DynamicVisit (Person p) { return Visit ((dynamic)p); }

  protected abstract T Visit (Person p);
  protected virtual T Visit (Customer c) { return Visit ((Person) c); }
  protected virtual T Visit (Employee e) { return Visit ((Person) e); }
}
```

Затем в подклассах не придется определять собственный метод `DynamicVisit`: они будут лишь переопределять версии метода `Visit`, поведение которых должно быть специализировано. Вдобавок появляются преимущества централизации методов, охватывающих иерархию `Person`, и возможность у реализующих классов вызывать базовые методы более естественным образом:

```
class ToXElementPersonVisitor : PersonVisitor<XElement>
{
  protected override XElement Visit (Person p)
  {
    return new XElement ("Person",
      new XAttribute ("Type", p.GetType().Name),
      new XElement ("FirstName", p.FirstName),
      new XElement ("LastName", p.LastName),
      p.Friends.Select (f => DynamicVisit (f))
    );
  }
```

```
  protected override XElement Visit (Customer c)
  {
    XElement xe = base.Visit (c);
    xe.Add (new XElement ("CreditLimit", c.CreditLimit));
    return xe;
  }
  protected override XElement Visit (Employee e)
  {
    XElement xe = base.Visit (e);
    xe.Add (new XElement ("Salary", e.Salary));
    return xe;
  }
}
```

В дальнейшем можно даже создавать подклассы самого класса `ToXElementPersonVisitor`.

## Множественная диспетчеризация

Язык C# и среда CLR всегда поддерживали ограниченную форму динамизма в виде вызовов виртуальных методов. Она отличается от динамического связывания C# тем, что для вызовов виртуальных методов компилятор должен фиксировать отдельный виртуальный член на этапе компиляции — основываясь на имени и сигнатуре вызываемого члена. Это означает, что справедливы приведенные ниже утверждения:

- выражение вызова должно полностью восприниматься компилятором (например, на этапе компиляции должно приниматься решение о том, чем является целевой член — полем или свойством);
- распознавание перегруженных членов должно осуществляться полностью компилятором на основе типов аргументов в течение этапа компиляции.

Следствие последнего утверждения заключается в том, что возможность выполнения вызовов виртуальных методов известна как *одиночная диспетчеризация*. Чтобы понять причину, взгляните на приведенный ниже вызов метода (где `Walk` — виртуальный метод):

```
animal.Walk (owner);
```

Принятие решения во время выполнения о том, какой метод `Walk` вызывать — класса `Dog` (собака) или класса `Cat` (кошка) — зависит только от типа *получателя*, т.е. `animal` (отсюда и "одиночная"). Если многочисленные перегруженные версии `Walk` принимают разные типы `owner`, тогда перегруженная версия выбирается на этапе компиляции безотносительно к тому, каким будет действительный тип объекта `owner` во время выполнения. Другими словами, только тип *получателя* во время выполнения может изменить то, какой метод будет вызван.

По контрасту динамический вызов откладывает распознавание перегруженных членов вплоть до времени выполнения:

```
animal.Walk ((dynamic) owner);
```

Окончательный выбор метода `Walk`, подлежащего вызову, теперь зависит и от `animal`, и от `owner` — это называется *множественной диспетчеризацией*, потому что в определении вызываемого метода `Walk` принимают участие не только тип получателя, но и типы аргументов времени выполнения.

## Анонимный вызов членов обобщенного типа

Строгость статической типизации C# – палка о двух концах. С одной стороны, она обеспечивает определенную степень корректности на этапе компиляции. С другой стороны, иногда она делает некоторые виды кода трудными в представлении или вовсе невозможными, и тогда приходится прибегать к рефлексии. В таких ситуациях динамическое связывание является более чистой и быстрой альтернативой рефлексии.

Примером может служить необходимость работы с объектом G<T>, где тип T неизвестен. Проиллюстрировать сказанное можно, определив следующий класс:

```
public class Foo<T> { public T Value; }
```

Предположим, что затем мы записываем метод, как показано ниже:

```
static void Write (object obj)
{
  if (obj is Foo<>)                                  // Недопустимо
    Console.WriteLine ((Foo<>) obj).Value);          // Недопустимо
}
```

Такой код не скомпилируется: члены *несвязанных* обобщенных типов вызывать нельзя.

Динамическое связывание предлагает два средства, с помощью которых можно обойти данную проблему. Первое из них – доступ к члену Value динамическим образом:

```
static void Write (dynamic obj)
{
  try { Console.WriteLine (obj.Value); }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
}
```

Здесь имеется (потенциальное) преимущество работы с любым объектом, который определяет поле или свойство Value. Тем не менее, осталась еще пара проблем. Во-первых, перехват исключения в подобной манере несколько запутан и неэффективен (к тому же отсутствует возможность заранее узнать у DLR, будет ли эта операция успешной). Во-вторых, такой подход не будет работать, если Foo является интерфейсом (скажем, IFoo<T>) и удовлетворено одно из следующих условий:

- член Value не реализован явно;
- тип, который реализует интерфейс IFoo<T>, недоступен (подробнее об этом позже).

Более удачное решение состоит в написании перегруженного вспомогательного метода по имени GetFooValue и его вызов с применением *динамического распознавания перегруженных членов*:

```
static void Write (dynamic obj)
{
  object result = GetFooValue (obj);
  if (result != null) Console.WriteLine (result);
}

static T GetFooValue<T> (Foo<T> foo) { return foo.Value; }
static object GetFooValue (object foo) { return null; }
```

Обратите внимание, что мы перегрузили метод GetFooValue с целью приема параметра object, который действует в качестве запасного варианта для любого типа.

Во время выполнения при вызове GetFooValue с динамическим аргументом динамический связыватель C# выберет наилучшую перегруженную версию. Если рассматриваемый объект не основан на Foo<T>, то вместо генерации исключения связыватель выберет перегруженную версию с параметром object.

Альтернатива предусматривает написание только первой перегруженной версии метода GetFooValue и последующий перехват исключения RuntimeBinderException. Преимущество такого подхода в том, что он различает случай, когда значение foo.Value равно null. Недостаток связан с появлением накладных расходов в плане производительности, которые обусловлены генерацией и перехватом исключения.

В главе 19 мы решали ту же самую проблему с интерфейсом, используя рефлексию — и прикладывали гораздо больше усилий (см. раздел "Анонимный вызов членов обобщенного интерфейса" в главе 19). Там рассматривался пример проектирования более мощной версии метода ToString, которая воспринимала бы объекты, реализующие IEnumerable и IGrouping<,>. Ниже приведен тот же пример, решенный более элегантно за счет динамического связывания:

```
static string GetGroupKey<TKey,TElement> (IGrouping<TKey,TElement> group)
{
  return "Group with key=" + group.Key + ": ";
}

static string GetGroupKey (object source) { return null; }

public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  if (value is string) return (string) value;
  if (value.GetType().IsPrimitive) return value.ToString();

  StringBuilder sb = new StringBuilder();

  string groupKey = GetGroupKey ((dynamic)value); //Динамическая диспетчеризация
  if (groupKey != null) sb.Append (groupKey);

  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append (ToStringEx (element) + " ");

  if (sb.Length == 0) sb.Append (value.ToString());

  return "\r\n" + sb.ToString();
}
```

Вот код в действии:

```
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c) ));
```

```
Group with key=x: x
Group with key=y: y y
Group with key=z: z z z
```

Обратите внимание, что для решения задачи мы применяли динамическое *распознавание перегруженных членов*. Если бы взамен мы поступили следующим образом:

```
dynamic d = value;
try { groupKey = d.Value); }
catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
```

то код потерпел бы неудачу, потому что LINQ-операция GroupBy возвращает тип, реализующий интерфейс IGrouping<,>, который сам по себе является внутренним и по этой причине недоступным:

```
internal class Grouping : IGrouping<TKey,TElement>, ...
{
  public TKey Key;
  ...
}
```

Хотя свойство Key объявлено как public, содержащий его класс ограничивает данное свойство до internal, делая доступным только через интерфейс IGrouping<,>. И как объяснялось в главе 4, при динамическом обращении к члену Value нет никакого способа сообщить среде DLR о необходимости привязки к указанному интерфейсу.

# Реализация динамических объектов

Объект может предоставить свою семантику привязки, реализуя интерфейс IDynamicMetaObjectProvider или более просто – создавая подкласс DynamicObject, который предлагает стандартную реализацию этого интерфейса. Мы кратко демонстрировали такой подход в главе 4 с помощью следующего примера:

```
static void Main()
{
  dynamic d = new Duck();
  d.Quack();    // Quack method was called (Вызван метод Quack)
  d.Waddle();   // Waddle method was called (Вызван метод Waddle)
}
public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

## DynamicObject

В предыдущем примере мы переопределили метод TryInvokeMember, который позволяет потребителю вызывать на динамическом объекте метод вроде Quack или Waddle. Класс DynamicObject открывает другие виртуальные методы, которые дают потребителям возможность использовать также и другие конструкции программирования. Ниже перечислены конструкции, имеющие представления в языке C#.

| Метод | Конструкция программирования |
|---|---|
| TryInvokeMember | Метод |
| TryGetMember, TrySetMember | Свойство или поле |
| TryGetIndex, TrySetIndex | Индексатор |
| TryUnaryOperation | Унарная операция, такая как ! |
| TryBinaryOperation | Бинарная операция, такая как == |
| TryConvert | Преобразование (приведение) в другой тип |
| TryInvoke | Вызов на самом объекте, например, d("foo") |

При успешном выполнении эти методы должны возвращать `true`. Если они возвращают `false`, тогда среда DLR будет прибегать к услугам связывателя языка в поиске подходящего члена в самом (подклассе) `DynamicObject`. В случае неудачи генерируется исключение `RuntimeBinderException`.

Мы можем продемонстрировать работу `TryGetMember` и `TrySetMember` с классом, который позволяет динамически получать доступ к атрибуту в объекте `XElement` (`System.Xml.Linq`):

```
static class XExtensions
{
  public static dynamic DynamicAttributes (this XElement e => new XWrapper (e);
  class XWrapper : DynamicObject
  {
    XElement _element;
    public XWrapper (XElement e) { _element = e; }
    public override bool TryGetMember (GetMemberBinder binder,
                                       out object result)
    {
      result = _element.Attribute (binder.Name).Value;
      return true;
    }
    public override bool TrySetMember (SetMemberBinder binder,
                                       object value)
    {
      _element.SetAttributeValue (binder.Name, value);
      return true;
    }
  }
}
```

А так он применяется:

```
XElement x = XElement.Parse (@"<Label Text=""Hello"" Id=""5""/>");
dynamic da = x.DynamicAttributes();
Console.WriteLine (da.Id);         // 5
da.Text = "Foo";
Console.WriteLine (x.ToString()); // <Label Text="Foo" Id="5" />
```

Следующий код выполняет аналогичное действие для интерфейса `System.Data.IDataRecord`, упрощая его использование средствами чтения данных:

```
public class DynamicReader : DynamicObject
{
  readonly IDataRecord _dataRecord;
  public DynamicReader (IDataRecord dr) { _dataRecord = dr; }
  public override bool TryGetMember (GetMemberBinder binder,
                                     out object result)
  {
    result = _dataRecord [binder.Name];
    return true;
  }
}
...
using (IDataReader reader = someDbCommand.ExecuteReader())
{
  dynamic dr = new DynamicReader (reader);
  while (reader.Read())
```

```
    {
      int id = dr.ID;
      string firstName = dr.FirstName;
      DateTime dob = dr.DateOfBirth;
      ...
    }
  }
```

В приведенном ниже коде демонстрируется работа `TryBinaryOperation` и `TryInvoke`:

```
  static void Main()
  {
    dynamic d = new Duck();
    Console.WriteLine (d + d);                    // foo
    Console.WriteLine (d (78, 'x'));              // 123
  }
  public class Duck : DynamicObject
  {
    public override bool TryBinaryOperation (BinaryOperationBinder binder,
                                             object arg, out object result)
    {
      Console.WriteLine (binder.Operation);   // Add
      result = "foo";
      return true;
    }
    public override bool TryInvoke (InvokeBinder binder,
                                    object[] args, out object result)
    {
      Console.WriteLine (args[0]);              // 78
      result = 123;
      return true;
    }
  }
```

Класс `DynamicObject` также открывает доступ к ряду виртуальных методов в интересах динамических языков. В частности, переопределение метода `GetDynamicMemberNames` позволяет возвращать список имен всех членов, которые предоставляет динамический объект.

Еще одна причина реализации `GetDynamicMemberNames` связана с тем, что отладчик Visual Studio задействует данный метод при отображении представления динамического объекта.

## ExpandoObject

Другое простое практическое применение `DynamicObject` касается написания динамического класса, который хранит и извлекает объекты в словаре с ключами-строками. Тем не менее, эта функциональность уже предлагается классом `ExpandoObject`:

```
dynamic x = new ExpandoObject();
x.FavoriteColor = ConsoleColor.Green;
x.FavoriteNumber = 7;
Console.WriteLine (x.FavoriteColor);          // Green
Console.WriteLine (x.FavoriteNumber);         // 7
```

Класс ExpandoObject реализует интерфейс IDictionary<string,object> и потому мы можем продолжить наш пример, как показано ниже:

```
var dict = (IDictionary<string,object>) x;
Console.WriteLine (dict ["FavoriteColor"]);      // Green
Console.WriteLine (dict ["FavoriteNumber"]);     // 7
Console.WriteLine (dict.Count);                  // 2
```

# Взаимодействие с динамическими языками

Хотя в языке C# поддерживается динамическое связывание через ключевое слово dynamic, оно не заходит настолько далеко, чтобы позволить оценивать выражение, описанное в строке, во время выполнения:

```
string expr = "2 * 3";
// "Выполнить" expr не удастся
```

Причина в том, что код для трансляции строки в дерево выражения требует лексического и семантического анализатора. Такие средства встроены в компилятор C# и не доступны в виде какой-то службы времени выполнения. Во время выполнения компилятор C# просто предоставляет *связыватель*, который сообщает среде DLR о том, как интерпретировать уже построенное дерево выражения.

Подлинные динамические языки, подобные IronPython и IronRuby, позволяют выполнять произвольную строку, что полезно при решении таких задач, как написание сценариев, динамическое конфигурирование и реализация процессоров динамических правил. Таким образом, хотя бо́льшую часть приложения можно написать на C#, для решения указанных задач удобно обращаться к какому-то динамическому языку. Кроме того, может возникнуть желание задействовать API-интерфейс, реализованный на динамическом языке, функциональность которого не имеет эквивалента в библиотеке .NET.

В следующем примере мы используем язык IronPython для оценки выражения, созданного во время выполнения, внутри кода C#. Данный сценарий мог бы применяться при написании калькулятора.

Чтобы запустить этот код, загрузите IronPython (воспользовавшись поисковой системой) и добавьте в свое приложение C# ссылки на сборки IronPython, Microsoft.Scripting и Microsoft.Scripting.Core.

```
using System;
using IronPython.Hosting;
using Microsoft.Scripting;
using Microsoft.Scripting.Hosting;
class Calculator
{
  static void Main()
  {
    int result = (int) Calculate ("2 * 3");
    Console.WriteLine (result);  // 6
  }
  static object Calculate (string expression)
  {
    ScriptEngine engine = Python.CreateEngine();
```

```
    return engine.Execute (expression);
  }
}
```

Поскольку мы передаем строку в Python, выражение будет оцениваться согласно правилам языка Python, а не C#. Это также означает возможность применения языковых средств Python, таких как списки:

```
var list = (IEnumerable) Calculate ("[1, 2, 3] + [4, 5]");
foreach (int n in list) Console.Write (n);  // 12345
```

## Передача состояния между C# и сценарием

Чтобы передать переменные из C# в Python, потребуется предпринять несколько дополнительных шагов, проиллюстрированных в следующем примере, который может служить основой для построения процессора правил:

```
// Следующая строка может поступать из файла или базы данных:
string auditRule = "taxPaidLastYear / taxPaidThisYear > 2";

ScriptEngine engine = Python.CreateEngine ();

ScriptScope scope = engine.CreateScope ();
scope.SetVariable ("taxPaidLastYear", 20000m);
scope.SetVariable ("taxPaidThisYear", 8000m);

ScriptSource source = engine.CreateScriptSourceFromString (
                              auditRule, SourceCodeKind.Expression);

bool auditRequired = (bool) source.Execute (scope);
Console.WriteLine (auditRequired);  // True
```

Вызвав метод `GetVariable`, переменные можно получить обратно:

```
string code = "result = input * 3";

ScriptEngine engine = Python.CreateEngine();

ScriptScope scope = engine.CreateScope();
scope.SetVariable ("input", 2);

ScriptSource source = engine.CreateScriptSourceFromString (code,
                              SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (engine.GetVariable (scope, "result"));  // 6
```

Обратите внимание, что во втором примере мы указали значение `SourceCodeKind.SingleStatement` (а не `Expression`), чтобы сообщить процессору о необходимости выполнения оператора.

Типы автоматически маршализируются между мирами .NET и Python. Можно даже обращаться к членам объектов .NET со стороны сценария:

```
string code = @"sb.Append (""World"")";

ScriptEngine engine = Python.CreateEngine ();

ScriptScope scope = engine.CreateScope ();
var sb = new StringBuilder ("Hello");
scope.SetVariable ("sb", sb);

ScriptSource source = engine.CreateScriptSourceFromString (
                              code, SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (sb.ToString());  // HelloWorld
```

# 21

# Безопасность

В настоящей главе мы обсудим два главных компонента, связанных с поддержкой безопасности в .NET:

- безопасность на основе удостоверений и ролей (авторизация);
- криптография.

Безопасность на основе удостоверений и ролей позволяет писать приложения, которые ограничивают то, *кто* именно и *что* конкретно может делать.

API-интерфейсы криптографии предназначены для хранения/обмена ценными данными, предотвращения пассивного перехвата данных, выявления подделки сообщений, генерации однонаправленных хешей с целью хранения паролей и создания цифровых подписей.

Мы также обсудим, как иметь дело безопасностью операционной системы (ОС), и унаследованную безопасность доступа кода (Code Access Security – CAS), ограничивающую операции, которые может выполнять код.

Типы, рассматриваемые в главе, определены в следующих пространствах имен:

```
System.Security;
System.Security.Permissions;
System.Security.Principal;
System.Security.Cryptography;
```

## Безопасность доступа кода

*Безопасность доступа кода* (CAS) позволяет среде CLR создавать среду с ограниченными возможностями, или *песочницу*, которая предотвращает выполнение кодом операций определенных видов (таких как чтение файлов ОС, проведение рефлексии или создание пользовательского интерфейса). Среду песочницы, созданную посредством CAS, называют средой с *частичным доверием*, тогда как нормальную неограниченную среду – средой с *полным доверием*.

Безопасность CAS считалась стратегически важной в начальный период существования .NET, т.к. она делала возможными следующие вещи:

- запуск элементов управления ActiveX в коде C# внутри веб-браузера (вроде аплетов Java);
- снижение стоимости совместного веб-хостинга за счет появления возможности выполнения внутри одного и того же процесса .NET множества веб-сайтов;
- развертывание приложений ClickOnce с ограниченными разрешениями через Интернет.

Первые две вещи больше не существенны, а третья всегда характеризовалась сомнительной ценностью, поскольку конечные пользователи вряд ли могут знать или понимать последствия назначения ограниченных разрешений перед установкой. И хотя имеются другие сценарии использования CAS, они более специализированы. Еще одна проблема в том, что песочница, созданная CAS, не является полностью надежной: в 2015 году в Microsoft заявили, что на CAS не следует полагаться как на механизм для обеспечения границ безопасности (и безопасность CAS была почти целиком исключена из стандарта .NET Standard 2.0). И это невзирая на усовершенствования CAS, внесенные в версии CLR 4 в 2010 году.

Организация песочниц, которые не опираются на CAS, по-прежнему актуальна: приложения UWP запускаются в песочнице, как и библиотеки CLR для работы с SQL Server. Такие песочницы навязываются ОС или размещающей системой CLR, они надежнее песочниц CAS, к тому же проще для понимания и управления. Безопасность ОС также работает с неуправляемым кодом, поэтому приложение UWP не может читать/записывать в произвольные файлы, будь оно реализовано на C# или C++.

По указанным причинам CAS в настоящей книге больше не рассматривается. Однако на веб-сайте издательства для загрузки доступны материалы из предыдущего издания. (Если вы — автор библиотек, тогда все еще можете нуждаться в обеспечении сред с частичным доверием в целях поддержки старых платформ.)

# Безопасность на основе удостоверений и ролей

Безопасность на основе удостоверений и ролей применяется для авторизации, обычно на сервере промежуточного уровня или в приложении ASP.NET: она позволяет ограничивать функциональность согласно имени аутентифицированного пользователя (удостоверение) или его роли (группа).

Безопасность на основе удостоверений и ролей полагается на *разрешения*, которые мы рассмотрим следующими.

## Разрешения

*Разрешение* действует в качестве шлюза, который условным образом предотвращает выполнение кода. Для авторизации мы используем класс `PrincipalPermission`, который описывает удостоверение и/или роль (например, "Mary" или "Human Resources"). Вот его конструктор:

```
public PrincipalPermission (string name, string role);
```

Важные методы `PrincipalPermission` определяются интерфейсом `IPermission`, которые он реализует:

```
public interface IPermission
{
  void Demand();
  IPermission Intersect (IPermission target);
  IPermission Union (IPermission target);
  bool IsSubsetOf (IPermission target);
  IPermission Copy();
}
```

Ключевым методом здесь является `Demand`. Он осуществляет выборочную проверку, чтобы выяснить, выдано ли в данный момент разрешение, и генерирует исключение `SecurityException`, если нет.

Например, чтобы обеспечить возможность запуска построения административных отчетов только пользователем Mary, можно написать следующий код:

```
new PrincipalPermission ("Mary", null).Demand();
// ... запустить построение административных отчетов
```

Методы Intersect и Union комбинируют два объекта разрешений одинакового типа в один. Цель метода Intersect заключается в создании "меньшего" объекта разрешения, тогда как цель Union – создание "большего" объекта разрешения.

В рамках разрешений на основе участников "больший" объект разрешения является *менее* ограничивающим при вызове Demand, т.к. для удовлетворения требования достаточно только *одного* из участников или удостоверений.

Метод IsSubsetOf возвращает true, если текущее разрешение является подмножеством заданного разрешения:

```
PrincipalPermission jay = new PrincipalPermission ("Jay", null);
PrincipalPermission sue = new PrincipalPermission ("Sue", null);

PrincipalPermission jayOrSue = (PrincipalPermission) jay.Union (sue);
Console.WriteLine (jay.IsSubsetOf (jayOrSue));  // True
```

В данном примере вызов метода Intersect на объектах jay и sue сгенерирует пустое разрешение, потому что они не перекрываются.

## Сравнение декларативной и императивной безопасности

До сих пор мы вручную создавали объекты разрешений и вызывали на них метод Demand. Это *императивная безопасность*. Того же самого результата можно достигнуть, добавляя атрибуты к методу, конструктору, классу, структуре или сборке – такой подход называется *декларативной безопасностью*. Хотя императивная безопасность обладает большей гибкостью, декларативная безопасность дает три преимущества:

- может означать написание меньшего объема кода;
- позволяет среде CLR заблаговременно определять, какие разрешения требует сборка;
- способна улучшить показатели производительности.

Вот пример:

```
[PrincipalPermission (SecurityAction.Demand, Name="Mary")]
public ReportData GetReports()
{
  ...
}
```

Прием работает, поскольку каждый тип разрешения имеет родственный тип атрибута в .NET Framework. Тип PrincipalPermission располагает родственным типом PrincipalPermissionAttribute. Первый аргумент конструктора атрибута всегда является значением перечисления SecurityAction, которое указывает, какой метод безопасности должен быть вызван после создания объекта разрешения (обычно Demand). Остальные именованные параметры отображаются на свойства соответствующего объекта разрешения.

## Реализация безопасности на основе удостоверений и ролей

В типовом сервере приложений разрешение `PrincipalPermission` запрашивается на всех методах, открытых клиенту, для которых необходимо обеспечить безопасность. Например, следующий метод требует, чтобы вызывающий его пользователь был членом роли `finance`:

```
[PrincipalPermission (SecurityAction.Demand, Role = "finance")]
public decimal GetGrossTurnover (int year)
{
  ...
}
```

Для обеспечения возможности вызова метода только отдельному пользователю взамен можно указать его имя в `Name`:

```
[PrincipalPermission (SecurityAction.Demand, Name = "sally")]
```

(Разумеется, необходимость жесткого кодирования имен делает код трудным для сопровождения.) Чтобы позволить комбинировать удостоверения и роли, взамен придется применять императивную безопасность. Это означает создание объектов `PrincipalPermission`, вызов `Union` для их объединения и затем вызов `Demand` на конечном результате.

## Назначение пользователей и ролей

Прежде чем запрос `PrincipalPermission` сможет достичь цели, объект реализации `IPrincipal` потребуется присоединить к текущему потоку.

Указать, что текущий пользователь Windows должен служить удостоверением, можно двумя способами, воздействующими на целый домен приложения либо только на текущий поток:

```
AppDomain.CurrentDomain.SetPrincipalPolicy (PrincipalPolicy.
                                            WindowsPrincipal);
```

или:

```
Thread.CurrentPrincipal = new WindowsPrincipal (WindowsIdentity.
                                                GetCurrent());
```

Если используется инфраструктура WCF или ASP.NET, то она способна помочь с воплощением удостоверения клиента. Это также можно делать самостоятельно с применением классов `GenericPrincipal` и `GenericIdentity`. Следующий код создает пользователя по имени `Jack` и назначает ему три роли:

```
GenericIdentity id = new GenericIdentity ("Jack");
GenericPrincipal p = new GenericPrincipal
  (id, new string[] { "accounts", "finance", "management" } );
```

Чтобы такое действие вступило в силу, объект удостоверения необходимо присоединить к текущему потоку:

```
Thread.CurrentPrincipal = p;
```

Удостоверение основано на потоке, потому что сервер приложений обычно обрабатывает множество клиентских запросов параллельно — каждый в собственном потоке. Поскольку каждый запрос может поступать из отличающегося клиента, он нуждается в другом удостоверении.

Допускается создавать подклассы классов `GenericIdentity` и `GenericPrincipal` либо реализовывать интерфейсы `IIdentity` и `IPrincipal` прямо в своих типах. Ниже приведены определения указанных интерфейсов:

```
public interface IIdentity
{
  string Name { get; }
  string AuthenticationType { get; }
  bool IsAuthenticated { get; }
}

public interface IPrincipal
{
  IIdentity Identity { get; }
  bool IsInRole (string role);
}
```

Ключевым методом является `IsInRole`. Обратите внимание, что здесь нет методов, возвращающих списки ролей, поэтому придется лишь проверять, действительна ли определенная роль для заданного удостоверения. Такая проверка может быть основой для построения более развитых систем авторизации.

# Подсистема безопасности операционной системы

Операционная система способна дополнительно ограничивать то, что может делать приложение, на основе привилегий учетной записи пользователя. В Windows существуют два типа учетных записей:

- административная учетная запись, на которую не накладываются ограничения в плане доступа к ресурсам локального компьютера;
- учетная запись с ограниченными разрешениями, которые сужают доступ к административным функциям и данным других пользователей.

Появившееся в версии Windows Vista средство, которое называется *контролем учетных записей пользователей* (User Account Control – UAC), предусматривает получение администраторами после входа в систему двух маркеров: административного и обычного пользователя. По умолчанию программы запускаются с маркером обычного пользователя, т.е. с ограниченными разрешениями, если только программа не требует повышения *полномочий до административных*. Затем пользователь должен санкционировать такой запрос в открывшемся диалоговом окне.

Для разработчиков приложений средство UAC означает, что *по умолчанию* приложение будет запускаться с ограниченными пользовательскими привилегиями. В результате вы должны выбрать один из двух подходов:

- строить приложение так, чтобы оно могло выполняться без административных привилегий;
- организовать в манифесте приложения требование повышения полномочий до административных.

Первый подход безопаснее и более удобен для пользователя. Проектирование программы для запуска без административных привилегий проще в большинстве случаев: ограничения гораздо менее "драконовские", чем в случае типичной песочницы CAS.

Выяснить, происходит ли выполнение от имени административной учетной записи, можно с помощью следующего метода:

```
[DllImport ("shell32.dll", EntryPoint = "#680")]
static extern bool IsUserAnAdmin();
```

При включенном средстве UAC метод возвращает `true`, только если текущий процесс имеет привилегии, повышенные до административных.

## Выполнение от имени учетной записи стандартного пользователя

Ниже перечислены ключевые действия, которые *нельзя* предпринимать при работе от имени учетной записи стандартного пользователя Windows:

- записывать в следующие каталоги:
- каталог операционной системы (обычно `\Windows`) и его подкаталоги;
- каталог файлов программ (`\Program Files`) и его подкаталоги;
- корневой каталог диска с операционной системой (например, `C:\`);
- записывать в ветвь `HKEY_LOCAL_MACHINE` реестра;
- читать данные мониторинга производительности (WMI).

Кроме того, как обычный пользователь (или даже администратор), вы можете получить отказ в доступе к файлам или ресурсам, которые принадлежат другим пользователям. Для защиты таких ресурсов в Windows используется система списков контроля доступа (Access Control List – ACL) – выдавать запросы и утверждения собственных прав в списках ACL можно через типы из пространства имен `System.Security.AccessControl`. Списки ACL могут также применяться к межпроцессным дескрипторам ожидания, описанным в главе 22.

В случае отказа в доступе к любому ресурсу среда CLR обнаруживает его и генерирует исключение `UnauthorizedAccessException` (а не молча терпит неудачу).

Безопасность доступа кода (CAS), которая теперь считается устаревшей, предоставляет еще один уровень защиты, обеспечиваемый средой CLR, а не ОС. Поскольку безопасность CAS и списки ACL независимы, вы можете успешно пройти систему безопасности CAS (например, запросив с помощью `Demand` разрешение `FileIOPermission`), но при попытке доступа к файлу все равно получить исключение `UnauthorizedAccessException` из-за ограничений ACL.

В большинстве случаев приходится учитывать ограничения стандартного пользователя следующим образом:

- выполнять запись в файлы в рекомендованных местоположениях;
- избегать использования реестра для хранения информации, которая может помещаться в файлы (помимо ветви `HKEY_CURRENT_USER`, к которой имеется доступ по чтению/записи);
- регистрировать компоненты ActiveX или COM во время установки.

Рекомендованным местоположением для пользовательских документов является `SpecialFolder.MyDocuments`:

```
string docsFolder = Environment.GetFolderPath
                    (Environment.SpecialFolder.MyDocuments);
string path = Path.Combine (docsFolder, "test.txt");
```

Рекомендованное местоположение для конфигурационных файлов, которые пользователь может модифицировать за пределами приложения, выглядит как `SpecialFolder.ApplicationData` (только текущий пользователь) или `SpecialFolder.CommonApplicationData` (все пользователи). Внутри указанных папок обычно создаются подкаталоги с именами, основанными на названиях организации и продукта.

## Повышение полномочий до административных и виртуализация

В главе 18 мы показали, как развертывать манифест приложения. С помощью манифеста приложения можно потребовать, чтобы при любом запуске вашей программы ОС Windows запрашивала у пользователя повышение полномочий до административных:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

Если вы замените `requireAdministrator` значением `asInvoker`, то оно сообщит ОС Windows о том, что повышение полномочий до административных *не* обязательно. Результат окажется почти таким же, как и при отсутствии манифеста приложения, но только отключается *виртуализация*. Виртуализация представляет собой временную меру, введенную в Windows Vista, которая помогает старым приложениям корректно запускаться без административных привилегий. Отсутствие манифеста приложения с элементом `requestedExecutionLevel` активизирует данное средство обратной совместимости.

Виртуализация вступает в игру, когда приложение осуществляет запись в каталог `Program Files` или `Windows` либо в ветвь `HKEY_LOCAL_MACHINE` реестра. Вместо генерации исключения изменения перенаправляются в отдельное местоположение на жестком диске, где они не могут оказать воздействие на исходные данные. Это предотвращает влияние приложения на ОС или на другие нормально функционирующие приложения.

# Обзор криптографии

В табл. 21.1 приведена сводка по возможностям криптографии в .NET. Мы кратко рассмотрим их в оставшихся разделах главы.

**Таблица 21.1. Возможности шифрования и хеширования в .NET**

| Возможность | Количество ключей, подлежащих управлению | Скорость | Прочность | Примечания |
|---|---|---|---|---|
| `File.Encrypt` | 0 | Высокая | Зависит от пароля пользователя | Защищает файлы прозрачным образом при поддержке со стороны файловой системы. Ключ неявно выводится из учетных данных вошедшего в систему пользователя |
| Защита данных Windows | 0 | Высокая | Зависит от пароля пользователя | Шифрует и расшифровывает байтовые массивы, используя неявно выведенный ключ |
| Хеширование | 0 | Высокая | Высокая | Однонаправленная (необратимая) трансформация. Применяется для хранения паролей, сравнения файлов и проверки данных на предмет разрушения |
| Симметричное шифрование | 1 | Высокая | Высокая | Для универсального шифрования/расшифровки. При шифровании и расшифровке используется один и тот же ключ. Может применяться для защиты сообщений при транспортировке |
| Шифрование с открытым ключом | 2 | Низкая | Высокая | Шифрование и расшифровка используют разные ключи. Применяется для обмена симметричным ключом во время передачи сообщений и для цифрового подписания файлов |

Инфраструктура .NET Framework также предлагает более специализированную поддержку для создания и проверки основанных на XML подписей в пространстве имен `System.Security.Cryptography.Xml` и типы для работы с цифровыми сертификатами в пространстве имен `System.Security.Cryptography.X509Certificates`.

# Защита данных Windows

В разделе "Операции с файлами и каталогами" главы 15 было показано, как использовать `File.Encrypt` для запрашивания у операционной системы прозрачного шифрования файла:

```
File.WriteAllText ("myfile.txt", "");
File.Encrypt ("myfile.txt");
File.AppendAllText ("myfile.txt", "sensitive data");
```

В данном случае шифрование применяет ключ, выведенный из пароля пользователя, который вошел в систему. Тот же самый неявно выведенный ключ можно использовать для шифрования байтового массива с помощью API-интерфейса защиты данных Windows (Windows Data Protection API). Интерфейс Data Protection API доступен через класс ProtectedData – простой тип с двумя статическими методами:

```
public static byte[] Protect (byte[] userData, byte[] optionalEntropy,
                              DataProtectionScope scope);
public static byte[] Unprotect (byte[] encryptedData, byte[] optionalEntropy,
                                DataProtectionScope scope);
```

Большинство типов из пространства имен System.Security.Cryptography находятся в сборках mscorlib.dll и System.dll. Исключением является класс ProtectedData: он расположен в сборке System.Security.dll.

Все, что вы включите в optionalEntropy, добавится к ключу, повысив тем самым его безопасность. Аргумент типа перечисления DataProtectionScope имеет два члена: CurrentUser и LocalMachine. В случае CurrentUser ключ выводится из учетных данных вошедшего в систему пользователя, а в случае LocalMachine применяется ключ уровня машины, общий для всех пользователей. Ключ LocalMachine обеспечивает меньшую защиту, но работает с Windows-службой или программой, которая должна функционировать под управлением множества учетных записей.

Ниже приведена простая демонстрация шифрования и расшифровки:

```
byte[] original = {1, 2, 3, 4, 5};
DataProtectionScope scope = DataProtectionScope.CurrentUser;

byte[] encrypted = ProtectedData.Protect (original, null, scope);
byte[] decrypted = ProtectedData.Unprotect (encrypted, null, scope);
// decrypted теперь содержит {1, 2, 3, 4, 5}
```

Защита данных Windows обеспечивает умеренное противодействие атакующему злоумышленнику, имеющему полный доступ к компьютеру, которое зависит от прочности пользовательского пароля. На уровне LocalMachine такая защита эффективна только против злоумышленников с ограниченным физическим и электронным доступом.

# Хеширование

Хеширование реализует однонаправленное шифрование. Оно идеально подходит для хранения паролей в базе данных, т.к. потребность в просмотре их расшифрованных версий может никогда не возникнуть. При аутентификации необходимо просто хешировать то, что ввел пользователь, и сравнивать с тем, что хранится в базе данных.

Независимо от длины исходных данных хеш-код всегда имеет небольшой фиксированный размер, что делает его удобным при сравнении файлов или обнаружении ошибок в потоке данных (довольно похоже на контрольную сумму). Изменение одного бита где-нибудь в исходных данных дает в результате значительно отличающийся хеш-код.

Для выполнения хеширования вызывается метод ComputeHash на одном из подклассов HashAlgorithm, таком как SHA256 или MD5:

```
byte[] hash;
using (Stream fs = File.OpenRead ("checkme.doc"))
  hash = MD5.Create().ComputeHash (fs);  // hash имеет длину 16 байтов
```

Метод ComputeHash также принимает байтовый массив, что удобно при хешировании паролей:

```
byte[] data = System.Text.Encoding.UTF8.GetBytes ("stRhong%pword");
byte[] hash = SHA256.Create().ComputeHash (data);
```

Метод GetBytes объекта Encoding преобразует строку в байтовый массив; метод GetString осуществляет обратное преобразование. Тем не менее, объект Encoding не может преобразовывать зашифрованный или хешированный байтовый массив в строку, потому что такие данные обычно нарушают правила кодирования текста. Взамен придется использовать методы Convert.ToBase64String и Convert.FromBase64String: они выполняют преобразования между любым байтовым массивом и допустимой (к тому же дружественной к XML) строкой.

MD5 и SHA256 — два подтипа HashAlgorithm, предоставленные .NET Framework. Вот все основные алгоритмы, расположенные в порядке возрастания степени безопасности (и длины хеша в байтах):

```
MD5(16) → SHA1(20) → SHA256(32) → SHA384(48) → SHA512(64)
```

Чем короче хеш, тем быстрее алгоритм выполняется. Алгоритм MD5 более чем в 20 раз быстрее алгоритма SHA512 и хорошо подходит для вычисления контрольных сумм файлов. С помощью MD5 можно хешировать сотни мегабайтов в секунду и затем сохранять результат в Guid. (Структура Guid имеет длину ровно 16 байтов и как тип значения проще в обработке, чем байтовый массив; скажем, значения Guid можно осмысленно сравнивать друг с другом посредством простой операции равенства.) Однако более короткие хеши увеличивают вероятность возникновения *коллизии* (когда два отличающихся файла дают один и тот же хеш).

При хешировании паролей и других чувствительных к безопасности данных применяйте, *по меньшей мере*, алгоритм SHA256. Алгоритмы MD5 и SHA1 для этих целей считаются ненадежными, и они подходят только для защиты от случайного повреждения данных, а не от их преднамеренной подделки.

Алгоритм SHA384 не быстрее SHA512, так что если требуется более высокая степень защиты, чем обеспечиваемая алгоритмом SHA256, то можно также использовать SHA512.

Более длинные алгоритмы SHA подходят для хеширования паролей, но требуют применения политики сильных паролей во избежание *словарной атаки* — стратегии, при которой злоумышленник строит таблицу поиска пароля путем хеширования каждого слова из словаря. Против этого можно обеспечить дополнительную защиту, "растягивая" хеши паролей, т.е. многократно производя повторное хеширование для получения байтовых последовательностей, которые требуют более интенсивных вычислений. Если выполнить повторное хеширование 100 раз, то словарная атака, которая иначе заняла бы месяц, может потребовать примерно 8 лет. Именно такой вид растяжения выполняют классы Rfc2898DeriveBytes и PasswordDeriveBytes.

Другой способ противодействия словарным атакам предусматривает введение "начального значения" — длинной последовательности байтов, которая первоначально получается через генератор случайных чисел и затем комбинируется с каждым паро-

лем перед его хешированием. Это усложняет работу злоумышленникам двумя путями: хеши требуют большего времени на вычисление и может отсутствовать доступ к байтам начального значения.

Инфраструктура .NET Framework также предоставляет 160-битный алгоритм хеширования `RIPEMD`, несколько превышающий `SHA1` в плане безопасности. Однако его реализация в .NET неэффективна и потому он выполняется даже медленнее алгоритма `SHA512`.

# Симметричное шифрование

При симметричном шифровании для шифрования и расшифровки используется один и тот же ключ. В .NET Framework предлагаются четыре алгоритма симметричного шифрования, главный из которых – алгоритм Рэндала (Rijndael). Алгоритм Рэндала является быстрым и надежным и имеет две реализации:

- класс `Rijndael`, который был доступен, начиная с версии .NET Framework 1.0;
- класс `Aes`, который появился в версии .NET Framework 3.5.

Указанные два класса в основном идентичны за исключением того, что `Aes` не позволяет ослаблять шифр, изменяя размер блока. Класс `Aes` рекомендуется к применению командой, отвечающей за безопасность CLR.

Классы `Rijndael` и `Aes` допускают использование симметричных ключей длиной 16, 24 или 32 байта: все они считаются безопасными. Ниже показано, как зашифровать последовательности байтов при записи их в файл с применением 16-байтового ключа:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv  = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};

byte[] data = { 1, 2, 3, 4, 5 };  // Данные, которые будут зашифрованы.

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
using (Stream f = File.Create ("encrypted.bin"))
using (Stream c = new CryptoStream (f, encryptor, CryptoStreamMode.Write))
  c.Write (data, 0, data.Length);
```

Следующий код расшифровывает содержимое этого файла:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv  = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};

byte[] decrypted = new byte[5];

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
using (Stream f = File.OpenRead ("encrypted.bin"))
using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
  for (int b; (b = c.ReadByte()) > -1;)
    Console.Write (b + " ");  // 1 2 3 4 5
```

В приведенном примере мы формируем ключ из 16 случайно выбранных байтов. Если при расшифровке указан неправильный ключ, тогда `CryptoStream` генерирует исключение `CryptographicException`. Перехват данного исключения – единственный способ проверки корректности ключа.

Помимо ключа мы строим *вектор инициализации* (Initialization Vector – IV). Такая 16-байтовая последовательность формирует часть шифра (почти как ключ), но не считается *секретной*. При передаче зашифрованного сообщения вектор IV можно отправ-

лять в виде простого текста (скажем, в заголовке сообщения) и затем *изменять в каждом сообщении*. Это сделает каждое зашифрованное сообщение нераспознаваемым на основе любых предшествующих сообщений, даже если их незашифрованные версии были похожими или идентичными.

Если защита посредством вектора IV не нужна или нежелательна, то ее можно аннулировать, используя одно и то же 16-байтовое значение для ключа и вектора IV. Тем не менее, отправка множества сообщений с одинаковым вектором IV ослабляет шифр и даже делает возможным его взлом.

Работа, связанная с криптографией, разделена между классами. Класс `Aes` решает математические задачи; он применяет алгоритм шифрования вместе с его объектами шифратора и дешифратора. Класс `CryptoStream` является связующим звеном; он заботится о взаимодействии с потоками. Класс `Aes` можно заменить другим классом симметричного алгоритма, но по-прежнему пользоваться `CryptoStream`.

Класс `CryptoStream` является *двунаправленным*, что означает возможность чтения или записи в поток в зависимости от выбора `CryptoStreamMode.Read` или `CryptoStreamMode.Write`. Шифратор и дешифратор умеют выполнять чтение *и* запись, давая в результате четыре комбинации. Чтение может быть удобно моделировать как "выталкивание", а запись — как "заталкивание". В случае сомнений начните с `Write` для шифрования и `Read` для расшифровки; зачастую такой подход наиболее естественен.

Для генерации случайного ключа или вектора IV применяйте класс `RandomNumberGenerator` из пространства имен `System.Cryptography`. Числа, которые он производит, являются по-настоящему непредсказуемыми, или *криптостойкими* (класс `System.Random` подобное не гарантирует). Вот пример:

```
byte[] key = new byte [16];
byte[] iv  = new byte [16];
RandomNumberGenerator rand = RandomNumberGenerator.Create();
rand.GetBytes (key);
rand.GetBytes (iv);
```

Если ключ и вектор IV не указаны, тогда криптостойкие случайные числа генерируются автоматически. Ключ и вектор IV можно получить через свойства `Key` и `IV` объекта `Aes`.

## Шифрование в памяти

С помощью класса `MemoryStream` шифрование и расшифровку можно производить полностью в памяти, для чего существуют вспомогательные методы, работающие с байтовыми массивами:

```
public static byte[] Encrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Aes algorithm = Aes.Create())
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
    return Crypt (data, encryptor);
}
public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Aes algorithm = Aes.Create())
  using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
    return Crypt (data, decryptor);
}
```

```
static byte[] Crypt (byte[] data, ICryptoTransform cryptor)
{
  MemoryStream m = new MemoryStream();
  using (Stream c = new CryptoStream (m, cryptor, CryptoStreamMode.Write))
    c.Write (data, 0, data.Length);
  return m.ToArray();
}
```

Здесь `CryptoStreamMode.Write` хорошо работает как для шифрования, так и для расшифровки, поскольку в обоих случаях осуществляется "заталкивание" внутрь нового потока в памяти.

Ниже приведены перегруженные версии методов, которые принимают и возвращают строки:

```
public static string Encrypt (string data, byte[] key, byte[] iv)
{
  return Convert.ToBase64String (
    Encrypt (Encoding.UTF8.GetBytes (data), key, iv));
}

public static string Decrypt (string data, byte[] key, byte[] iv)
{
  return Encoding.UTF8.GetString (
    Decrypt (Convert.FromBase64String (data), key, iv));
}
```

В следующем коде демонстрируется их использование:

```
byte[] kiv = new byte[16];
RandomNumberGenerator.Create().GetBytes (kiv);

string encrypted = Encrypt ("Yeah!", kiv, kiv);
Console.WriteLine (encrypted);  // R1/5gYvcxyR2vzPjnT7yaQ==

string decrypted = Decrypt (encrypted, kiv, kiv);
Console.WriteLine (decrypted);  // Yeah!
```

## Соединение в цепочку потоков шифрования

Класс `CryptoStream` представляет собой декоратор, означая возможность соединения в цепочки с другими потоками. В показанном ниже примере мы записываем сжатый зашифрованный текст в файл, после чего читаем его обратно:

```
// Использовать при демонстрации стандартный ключ/IV.
using (Aes algorithm = Aes.Create())
{
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor())
  using (Stream f = File.Create ("serious.bin"))
  using (Stream c = new CryptoStream (f,encryptor,CryptoStreamMode.Write))
  using (Stream d = new DeflateStream (c, CompressionMode.Compress))
  using (StreamWriter w = new StreamWriter (d))
    await w.WriteLineAsync ("Small and secure!");

  using (ICryptoTransform decryptor = algorithm.CreateDecryptor())
  using (Stream f = File.OpenRead ("serious.bin"))
  using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
  using (Stream d = new DeflateStream (c, CompressionMode.Decompress))
  using (StreamReader r = new StreamReader (d))
    Console.WriteLine (await r.ReadLineAsync());  // Small and secure!
}
```

(В качестве финального штриха мы сделали программу асинхронной, вызывая методы `WriteLineAsync` и `ReadLineAsync`, а затем ожидая результат.)

В данном примере все однобуквенные переменные формируют часть цепочки. Объекты `algorithm`, `encryptor` и `decryptor` помогают `CryptoStream` выполнять работу по шифрованию. На рис. 21.1 приведена соответствующая диаграмма.

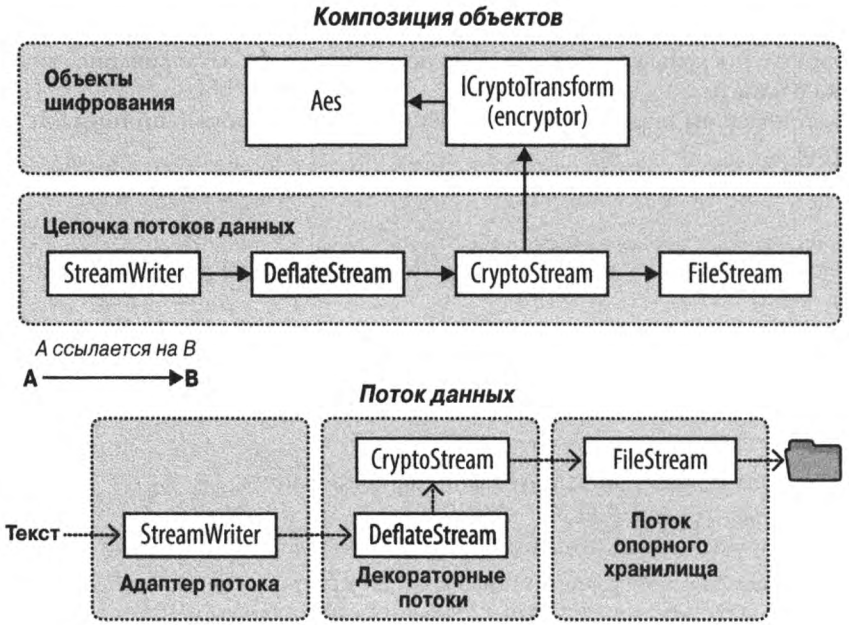



***Рис. 21.1.*** *Соединение в цепочку потоков шифрования и сжатия*

Соединение в цепочку потоков в такой манере требует мало памяти вне зависимости от конечных размеров потоков.

Вместо вложения множества операторов `using` друг в друга цепочку можно сконструировать следующим образом:

```
using (ICryptoTransform encryptor = algorithm.CreateEncryptor())
using
  (StreamWriter w = new StreamWriter (
    new DeflateStream (
      new CryptoStream (
        File.Create ("serious.bin"),
        encryptor,
        CryptoStreamMode.Write
      ),
      CompressionMode.Compress)
    )
  )
```

Однако такой подход менее надежен, чем предыдущий, поскольку в случае генерации исключения в конструкторе какого-нибудь объекта (например, `DeflateStream`) любые уже созданные объекты (скажем, `FileStream`) не будут освобождены.

## Освобождение объектов шифрования

Освобождение объекта `CryptoStream` гарантирует, что содержимое его внутреннего кеша данных будет сброшено в лежащий в основе поток. Внутреннее кеширование является необходимым для алгоритмов шифрования, т.к. они обрабатывают данные блоками, а не по одному байту за раз.

Класс `CryptoStream` необычен тем, что его метод `Flush` ничего не делает. Чтобы сбросить поток (не освобождая его), потребуется вызвать метод `FlushFinalBlock`. В противоположность методу `Flush` метод `FlushFinalBlock` может быть вызван только однократно, после чего никакие дополнительные данные записать не удастся.

В рассматриваемых примерах мы также освобождаем объект `Aes` и объекты, реализующие `ICryptoTransform` (`encryptor` и `decryptor`). На самом деле освобождение в случае применения алгоритма Рэндала не обязательно, потому что все его реализации являются управляемыми. Тем не менее, освобождение по-прежнему играет важную роль: в памяти очищается симметричный ключ и связанные с ним данные, предотвращая последующее их обнаружение другим программным обеспечением, которое функционирует на компьютере (речь идет о вредоносном ПО). В выполнении этой работы нельзя полагаться на сборщик мусора, поскольку он просто помечает разделы памяти как свободные, не обнуляя все их байты.

Простейший способ освободить объект `Aes` за пределами оператора `using` – вызвать метод `Clear`. Его метод `Dispose` сокрыт через явную реализацию (чтобы сигнализировать о необычной семантике освобождения).

## Управление ключами

Жестко кодировать ключи шифрования не рекомендуется, т.к. сборки довольно легко декомпилировать, используя ряд популярных инструментов. Более удачное решение предусматривает построение для каждой установки случайного ключа и его сохранение безопасным образом с помощью защиты данных Windows (или путем шифрования всего сообщения посредством защиты данных Windows). Если производится шифрование потока сообщений, тогда шифрование с открытым ключом обеспечивает еще лучший вариант.

# Шифрование с открытым ключом и подписание

Криптография с открытым ключом является *асимметричной*, что означает применение разных ключей для шифрования и расшифровки.

В отличие от симметричного шифрования, где ключом может служить любая произвольная последовательность байтов подходящей длины, асимметричная криптография требует специально сформированных пар ключей. Пара ключей содержит компоненты *открытого ключа* и *секретного ключа*, которые работают вместе следующим образом:

- открытый ключ шифрует сообщения;
- секретный ключ расшифровывает сообщения.

Участник, "формирующий" пару ключей, хранит секретный ключ вдали от глаз, а открытый ключ распространяет свободно. Особенность такого типа криптографии заключается в том, что вычислить секретный ключ на основе открытого ключа невозможно. Таким образом, в случае утери секретного ключа зашифрованные с его помощью данные не могут быть восстановлены; если же произошла утечка секретного ключа, тогда вся система шифрования становится бесполезной.

Предоставление открытого ключа позволяет двум компьютерам взаимодействовать защищенным образом через публичную сеть без предварительного контакта и без существующего общего секрета. Чтобы посмотреть, как это работает, предположим, что компьютер *Origin* должен отправить конфиденциальное сообщение компьютеру *Target*.

1. Компьютер *Target* генерирует пару открытого и секретного ключей и затем отправляет открытый ключ компьютеру *Origin*.
2. Компьютер *Origin* шифрует конфиденциальное сообщение с использованием открытого ключа компьютера *Target*, после чего отправляет его *Target*.
3. Компьютер *Target* расшифровывает конфиденциальное сообщение с помощью своего секретного ключа.

А вот что будет видеть перехватчик:

- открытый ключ компьютера Target;
- конфиденциальное сообщение, зашифрованное посредством открытого ключа компьютера Target.

Однако без секретного ключа компьютера *Target* сообщение не может быть расшифровано.

Шифрование с открытым ключом не предотвращает атаки типа "человек посередине": другими словами, компьютер *Origin* не может знать, является компьютер *Target* злоумышленным участником или нет. Для аутентификации получателя отправитель уже должен знать открытый ключ получателя либо иметь возможность проверить достоверность ключа через *цифровой сертификат сайта*.

Конфиденциальное сообщение, отправленное из компьютера *Origin* в компьютер *Target*, обычно содержит новый ключ для последующего *симметричного* шифрования. Это позволяет прекратить шифрование с открытым ключом для оставшейся части сеанса и отдать предпочтение симметричному алгоритму, способному обрабатывать более крупные сообщения. Такой протокол особенно безопасен, если для каждого сеанса генерируется новая пара открытого и секретного ключей, так что хранить какие-либо ключи ни на том, ни на другом компьютере не понадобится.

Алгоритмы шифрования с открытым ключом полагаются на то, что сообщение по размерам меньше ключа. В итоге они становятся подходящими для шифрования только небольших объемов данных, таких как ключ для последующего симметричного шифрования. Если вы попытаетесь зашифровать сообщение, которое намного больше половины размера ключа, тогда поставщик криптографии сгенерирует исключение.

## Класс RSA

Инфраструктура .NET Framework предлагает несколько асимметричных алгоритмов, самым популярным из которых является RSA. Ниже показано, как шифровать и расшифровывать с помощью RSA:

```
byte[] data = { 1, 2, 3, 4, 5 };  // Это данные, которые будут шифроваться.
using (var rsa = new RSACryptoServiceProvider())
{
  byte[] encrypted = rsa.Encrypt (data, true);
  byte[] decrypted = rsa.Decrypt (encrypted, true);
}
```

Поскольку мы не указали открытый или секретный ключ, поставщик криптографии автоматически генерирует пару ключей, применяя стандартную длину 1024 бита; посредством конструктора можно запросить более длинные ключи с приращением в 8 байтов. Для приложений, критических в отношении безопасности, разумно запрашивать длину в 2048 бит:

```
var rsa = new RSACryptoServiceProvider (2048);
```

Генерация пары ключей связана с интенсивными вычислениями, требуя примерно 100 миллисекунд. По указанной причине реализация RSA задерживает генерацию вплоть до момента, когда ключ действительно необходим, скажем, при вызове метода `Encrypt`. Это дает шанс загрузить существующий ключ или пару ключей, если она существует.

Методы `ImportCspBlob` и `ExportCspBlob` загружают и сохраняют ключи в формате байтового массива. Методы `FromXmlString` и `ToXmlString` делают то же самое в формате строки, содержащей XML-фрагмент. Флаг `bool` позволяет указывать, нужно ли при сохранении включать секретный ключ. Ниже демонстрируется построение пары ключей и сохранение ее на диске:

```
using (var rsa = new RSACryptoServiceProvider())
{
  File.WriteAllText ("PublicKeyOnly.xml", rsa.ToXmlString (false));
  File.WriteAllText ("PublicPrivate.xml", rsa.ToXmlString (true));
}
```

Так как мы не предоставили существующие ключи, метод `ToXmlString` создаст новую пару ключей (при первом вызове). В следующем примере мы читаем эти ключи и используем их для шифрования и расшифровки сообщения:

```
byte[] data = Encoding.UTF8.GetBytes ("Message to encrypt");

string publicKeyOnly = File.ReadAllText ("PublicKeyOnly.xml");
string publicPrivate = File.ReadAllText ("PublicPrivate.xml");

byte[] encrypted, decrypted;

using (var rsaPublicOnly = new RSACryptoServiceProvider())
{
  rsaPublicOnly.FromXmlString (publicKeyOnly);
  encrypted = rsaPublicOnly.Encrypt (data, true);

  // Следующая строка кода сгенерирует исключение, потому что
  // для расшифровки необходим секретный ключ:
  // decrypted = rsaPublicOnly.Decrypt (encrypted, true);
}

using (var rsaPublicPrivate = new RSACryptoServiceProvider())
{
  // С помощью секретного ключа можно успешно расшифровывать:
  rsaPublicPrivate.FromXmlString (publicPrivate);
  decrypted = rsaPublicPrivate.Decrypt (encrypted, true);
}
```

## Цифровые подписи

Алгоритмы с открытым ключом могут также применяться для цифрового подписания сообщений или документов. Подпись подобна хешу за исключением того, что ее создание требует секретного ключа, а потому она не может быть подделана. Для проверки подлинности подписи используется открытый ключ. Ниже приведен пример:

```
byte[] data = Encoding.UTF8.GetBytes ("Message to sign");
byte[] publicKey;
byte[] signature;
object hasher = SHA1.Create();  // Выбранный алгоритм хеширования.

// Сгенерировать новую пару ключей, затем с ее помощью подписать данные:
using (var publicPrivate = new RSACryptoServiceProvider())
{
  signature = publicPrivate.SignData (data, hasher);
  publicKey = publicPrivate.ExportCspBlob (false); // Получить открытый ключ.
}

// Создать новый объект поставщика шифрования RSA, используя
// только открытый ключ, затем протестировать подпись.
using (var publicOnly = new RSACryptoServiceProvider())
{
  publicOnly.ImportCspBlob (publicKey);
  Console.Write (publicOnly.VerifyData (data, hasher, signature));  // True

  // Давайте теперь подделаем данные и перепроверим подпись:
  data[0] = 0;
  Console.Write (publicOnly.VerifyData (data, hasher, signature));  // False

  // Следующий вызов генерирует исключение из-за отсутствия секретного ключа:
  signature = publicOnly.SignData (data, hasher);
}
```

Подписание работает за счет хеширования данных с последующим применением к результирующему хешу асимметричного алгоритма. Из-за того, что хеши имеют небольшой фиксированный размер, крупные документы могут подписываться относительно быстро (шифрование с открытым ключом намного интенсивнее эксплуатирует центральный процессор, чем хеширование). При желании можно выполнить хеширование самостоятельно, а затем вызвать метод `SignHash` вместо `SignData`:

```
using (var rsa = new RSACryptoServiceProvider())
{
  byte[] hash = SHA1.Create().ComputeHash (data);
  signature = rsa.SignHash (hash, CryptoConfig.MapNameToOID ("SHA1"));
  ...
}
```

Методу `SignHash` по-прежнему необходимо знать, какой алгоритм хеширования использовался; метод `CryptoConfig.MapNameToOID` предоставляет эту информацию в корректном формате на основе дружественного имени, такого как `"SHA1"`.

Класс `RSACryptoServiceProvider` генерирует подписи, размер которых соответствует размеру ключа. В настоящее время ни один из основных алгоритмов не генерирует защищенные подписи, длина которых была бы значительно меньше 128 байтов (подходящие, например, для кодов активации продуктов).

Чтобы подписание было эффективным, получатель должен знать и доверять открытому ключу отправителя, что можно обеспечить через заблаговременные коммуникации, предварительную конфигурацию или сертификат сайта. Сертификат сайта представляет собой электронную запись открытого ключа и имени отправителя, которая сама подписана независимым доверенным центром. Типы для работы с сертификатами определены в пространстве имен `System.Security.Cryptography.X509Certificates`.

# 22

# Расширенная многопоточность

Мы начинали главу 14 с рассмотрения основ многопоточности в качестве подготовки к исследованию задач и асинхронности. В частности, мы показали, каким образом стартовать/конфигурировать поток, и раскрыли такие основные концепции, как организация пула потоков, блокировка, зацикливание и контексты синхронизации. Мы также рассказали о блокировке и безопасности к потокам и продемонстрировали простейшую сигнализирующую конструкцию `ManualResetEvent`.

В настоящей главе мы возвращаемся к теме многопоточности. В первых трех разделах мы предоставим дополнительные сведения по синхронизации, блокировке и безопасности в отношении потоков. Затем мы рассмотрим следующие темы:

- немонопольное блокирование (`Semaphore` и блокировки объектов чтения/записи);
- все сигнализирующие конструкции (`AutoResetEvent`, `ManualResetEvent`, `CountdownEvent` и `Barrier`);
- ленивая инициализация (`Lazy<T>` и `LazyInitializer`);
- локальное хранилище потока (`ThreadStaticAttribute`, `ThreadLocal<T>` и `GetData`/`SetData`);
- вытесняющие многопоточные методы (`Interrupt`, `Abort`, `Suspend` и `Resume`);
- таймеры.

Многопоточность является настолько обширной темой, что у себя на веб-сайте по адресу `http://albahari.com/threading/` мы разместили дополнительные материалы, посвященные перечисленным ниже более тонким темам:

- использование методов `Monitor.Wait` и `Monitor.Pulse` в специализированных сигнализирующих сценариях;
- приемы неблокирующей синхронизации для микро-оптимизации (`Interlocked`, барьеры памяти, `volatile`);
- применение типов `SpinLock` и `SpinWait` в сценариях с высоким уровнем параллелизма.

# Обзор синхронизации

*Синхронизация* представляет собой акт координации параллельно выполняемых действий с целью получения предсказуемых результатов. Синхронизация особенно важна, когда множество потоков получают доступ к одним и тем же данным; в этой области удивительно легко столкнуться с серьезными трудностями.

Вероятно, простейшими и наиболее удобными инструментами синхронизации считаются продолжения и комбинаторы задач, описанные в главе 14. За счет представления параллельных программ в виде асинхронных операций, связанных вместе продолжениями и комбинаторами, уменьшается необходимость в блокировке и сигнализации. Тем не менее, по-прежнему встречаются ситуации, когда в игру вступают низкоуровневые конструкции.

Конструкции синхронизации могут быть разделены на три описанные ниже категории.

**Монопольное блокирование**

Конструкции монопольного блокирования позволяют выполнять некоторое действие или запускать определенный раздел кода только одному потоку в каждый момент времени. Их основное назначение заключается в том, чтобы предоставить потокам возможность доступа к записываемому разделяемому состоянию, не влияя друг на друга. Конструкциями монопольного блокирования являются `lock`, `Mutex` и `SpinLock`.

**Немонопольное блокирование**

Немонопольное блокирование позволяет *ограничивать* параллелизм. Конструкциями немонопольного блокирования являются `Semaphore` (`SemaphoreSlim`) и `ReaderWriterLock` (`ReaderWriterLockSlim`).

**Сигнализация**

Сигнализация позволяет потоку блокироваться вплоть до получения одного или большего числа уведомлений от другого потока (потоков). Сигнализирующие конструкции включают `ManualResetEvent` (`ManualResetEventSlim`), `AutoResetEvent`, `CountdownEvent` и `Barrier`. Первые три конструкции называются *дескрипторами ожидания событий.*

Кроме того, возможно (хотя и сложно) выполнять определенные параллельные операции на разделяемом состоянии без блокирования за счет использования *неблокирующих конструкций синхронизации.* Существуют методы `Thread.MemoryBarrier`, `Thread.VolatileRead` и `Thread.VolatileWrite`, ключевое слово `volatile`, а также класс `Interlocked`. Мы раскрываем эту тему на нашем веб-сайте вместе с методами `Wait/Pulse` класса `Monitor`, которые могут применяться для написания специальной сигнализирующей логики (`http://albahari.com/threading/`).

# Монопольное блокирование

Доступны три конструкции монопольного блокирования: оператор `lock`, класс `Mutex` и структура `SpinLock`. Конструкция `lock` является наиболее удобной и часто используемой, в то время как другие две конструкции ориентированы на собственные сценарии:

- класс Mutex позволяет охватывать множество процессов (блокировки на уровне компьютера);
- структура SpinLock реализует микро-оптимизацию, которая может уменьшить количество переключений контекста в сценариях с высоким уровнем параллелизма (http://albahari.com/threading/).

## Оператор lock

Для иллюстрации потребности в блокировании рассмотрим следующий класс:

```
class ThreadUnsafe
{
  static int _val1 = 1, _val2 = 1;

  static void Go()
  {
    if (_val2 != 0) Console.WriteLine (_val1 / _val2);
    _val2 = 0;
  }
}
```

Класс ThreadUnsafe не безопасен в отношении потоков: если метод Go был вызван двумя потоками одновременно, то появляется возможность получения ошибки деления на ноль. Дело в том, что в одном потоке поле _val2 может быть установлено в 0 как раз тогда, когда выполнение в другом потоке находится между оператором if и вызовом метода Console.WriteLine. Ниже показано, как исправить проблему с помощью lock:

```
class ThreadSafe
{
  static readonly object _locker = new object();
  static int _val1 = 1, _val2 = 1;

  static void Go()
  {
    lock (_locker)
    {
      if (_val2 != 0) Console.WriteLine (_val1 / _val2);
      _val2 = 0;
    }
  }
}
```

В каждый момент времени блокировать объект синхронизации (в данном случае _locker) может только один поток, и любые соперничающие потоки задерживаются до тех пор, пока блокировка не будет освобождена. Если за блокировку соперничают несколько потоков, тогда они ставятся в "очередь готовности" с предоставлением блокировки на основе "первым пришел – первым обслужен"[1]. Говорят, что монопольные блокировки иногда приводят к *последовательному* доступу к объекту, защищаемому блокировкой, т.к. доступ одного потока не может совмещаться с доступом другого. В рассматриваемом случае мы защищаем логику внутри метода Go, а также поля _val1 и _val2.

[1] Равноправие в этой очереди временами может нарушаться из-за нюансов поведения Windows и CLR.

## Monitor.Enter и Monitor.Exit

Фактически оператор lock в C# является синтаксическим сокращением для вызова методов Monitor.Enter и Monitor.Exit с добавленным блоком try/finally. Ниже показана упрощенная версия того, что на самом деле происходит внутри метода Go из предыдущего примера:

```
Monitor.Enter (_locker);
try
{
  if (_val2 != 0) Console.WriteLine (_val1 / _val2);
  _val2 = 0;
}
finally { Monitor.Exit (_locker); }
```

Вызов метода Monitor.Exit без предварительного вызова Monitor.Enter на том же объекте приводит к генерации исключения.

### Перегруженная версия Monitor.Enter, принимающая аргумент lockTaken

Продемонстрированный выше код представляет собой в точности то, что генерировали компиляторы версий C# 1.0, C# 2.0 и C# 3.0 при трансляции оператора lock.

Однако в данном коде присутствует тонкая уязвимость. Представим себе (маловероятный) случай генерации исключения между вызовом метода Monitor.Enter и блоком try (возможно, из-за вызова метода Abort на этом потоке либо выдачи исключения OutOfMemoryException). В таком сценарии блокировка может быть как получена, так и нет. Если блокировка *получена*, то она не будет освобождена, поскольку мы никогда не войдем в блок try/finally. В результате происходит утечка блокировки. Во избежание подобной опасности проектировщики CLR 4.0 добавили следующую перегруженную версию Monitor.Enter:

```
public static void Enter (object obj, ref bool lockTaken);
```

Значение lockTaken станет равно false, если (и только если) метод Enter сгенерировал исключение, и блокировка не была получена.

Вот как выглядит более надежный шаблон применения (именно так транслирует оператор lock компилятор C# 4.0 и последующих версий):

```
bool lockTaken = false;
try
{
  Monitor.Enter (_locker, ref lockTaken);
  // Выполнить необходимые действия...
}
finally { if (lockTaken) Monitor.Exit (_locker); }
```

### TryEnter

Класс Monitor также предлагает метод TryEnter, который позволяет указать тайм-аут в миллисекундах или в виде структуры TimeSpan. Метод TryEnter возвращает true, если блокировка получена, или false, если никаких блокировок не получено из-за истечения времени тайм-аута. Метод TryEnter можно также вызывать без аргументов, что дает возможность "проверить" блокировку, немедленно инициируя тайм-аут, если блокировка не может быть получена сразу. Как и Enter, в версии CLR 4.0 метод TryEnter перегружен для приема аргумента lockTaken.

## Выбор объекта синхронизации

Использовать в качестве объекта синхронизации можно любой объект, видимый каждому участвующему потоку, при одном жестком условии: он должен быть ссылочного типа. Объект синхронизации обычно является закрытым (потому что это помогает инкапсулировать логику блокирования) и, как правило, представляет собой поле экземпляра или статическое поле. Объект синхронизации может дублировать защищаемый посредством него объект, что делает поле `_list` в следующем примере:

```
class ThreadSafe
{
  List <string> _list = new List <string>();

  void Test()
  {
    lock (_list)
    {
      _list.Add ("Item 1");
      ...
```

Поле, выделенное для целей блокирования (вроде `_locker` в предыдущем примере), обеспечивает точный контроль над областью видимости и степенью детализации блокировки. Применяться в качестве объекта синхронизации может также содержащий объект (`this`) либо даже его тип:

```
lock (this) { ... }
```

или:

```
lock (typeof (Widget)) { ... }  // Для защиты доступа к статическим членам
```

Недостаток блокирования подобным образом связан с тем, что логика блокирования не инкапсулирована, а потому предотвратить взаимоблокировки и избыточные блокировки становится труднее. Блокировка на типе может также выходить за границы домена приложения (внутри одного и того же процесса, как объясняется в главе 24).

Можно также блокировать локальные переменные, захваченные лямбда-выражениями или анонимными методами.

Блокирование никак не ограничивает доступ к самому объекту синхронизации. Другими словами, вызов `x.ToString()` не будет блокироваться из-за того, что другой поток вызывает `lock(x)`; чтобы блокирование произошло, вызывать `lock(x)` должны оба потока.

## Когда нужна блокировка

Запомните базовое правило: блокировка необходима при доступе к *любому записываемому разделяемому полю*. Синхронизация должна приниматься во внимание даже в таком простейшем случае, как операция присваивания для одиночного поля. В следующем классе ни метод `Increment`, ни метод `Assign` не является безопасным в отношении потоков:

```
class ThreadUnsafe
{
  static int _x;
  static void Increment() { _x++; }
  static void Assign()    { _x = 123; }
}
```

А вот безопасные к потокам версии методов Increment и Assign:

```
static readonly object _locker = new object();
static int _x;

static void Increment() { lock (_locker) _x++; }
static void Assign()    { lock (_locker) _x = 123; }
```

Когда блокировки отсутствуют, могут возникнуть две проблемы.

- Операции вроде инкрементирования значения переменной (а в определенных обстоятельствах даже чтение/запись переменной) не являются атомарными.
- Компилятор, среда CLR и процессор имеют право изменять порядок следования инструкций и кешировать переменные в регистрах центрального процессора в целях улучшения производительности – до тех пор, пока такие оптимизации не изменяют поведение однопоточной программы (или многопоточной программы, в которой используются блокировки).

Блокирование смягчает вторую проблему, т.к. оно создает *барьер памяти* до и после блокировки. Барьер памяти представляет собой "заграждающую метку", которую не могут пересечь указанные эффекты либо изменение порядка следования и кеширование.

Сказанное применимо не только к блокировкам, но и ко всем конструкциям синхронизации. Таким образом, если используется, например, *сигнализирующая* конструкция, которая гарантирует, что в каждый момент времени переменная читается/записывается только одним потоком, тогда блокировка не нужна. Следовательно, показанный ниже код является безопасным к потокам без блокирования x:

```
var signal = new ManualResetEvent (false);
int x = 0;
new Thread (() => { x++; signal.Set(); }).Start();
signal.WaitOne();
Console.WriteLine (x);    // 1 (всегда)
```

В разделе "Nonblocking Synchronization" ("Неблокирующая синхронизация") на веб-сайте http://albahari.com/threading мы объясняем, как возникла такая потребность, и показываем, каким образом барьеры памяти и класс Interlocked могут предложить альтернативы блокированию в подобных ситуациях.

## Блокирование и атомарность

Если группа переменных всегда читается и записывается внутри одной и той же блокировки, то можно говорить о том, что эти переменные читаются и записываются *атомарно*. Давайте предположим, что поля x и y всегда читаются и устанавливаются внутри блокировки на объекте locker:

```
lock (locker) { if (x != 0) y /= x; }
```

Можно сказать, что доступ к x и y производится атомарно, поскольку блок кода не может быть разделен или вытеснен действиями другого потока так, что он изменит содержимое x или y и *сделает результаты недействительными*. Обеспечивая доступ к x и y всегда внутри одной и той же монопольной блокировки, вы никогда не получите ошибку деления на ноль.

Атомарность, обеспечиваемая блокировкой, нарушается, если внутри блока `lock` генерируется исключение. Например, взгляните на следующий код:

```
decimal _savingsBalance, _checkBalance;

void Transfer (decimal amount)
{
  lock (_locker)
  {
    _savingsBalance += amount;
    _checkBalance -= amount + GetBankFee();
  }
}
```

Если метод `GetBankFee` сгенерирует исключение, тогда банк потеряет деньги. В таком случае мы могли бы избежать проблемы, вызвав `GetBankFee` раньше. Решение для более сложных ситуаций предусматривает реализацию логики "отката" внутри блока `catch` или `finally`.

*Атомарность* инструкций представляет собой другую, хотя и похожую концепцию: инструкция считается атомарной, если она выполняется неделимым образом на лежащем в основе процессоре.

## Вложенное блокирование

Поток может многократно блокировать один и тот же объект вложенным (*реентерабельным*) образом:

```
lock (locker)
  lock (locker)
    lock (locker)
    {
      // Выполнить необходимые действия...
    }
```

или:

```
Monitor.Enter (locker); Monitor.Enter (locker); Monitor.Enter (locker);
// Выполнить необходимые действия...
Monitor.Exit (locker);  Monitor.Exit (locker);  Monitor.Exit (locker);
```

В таких сценариях объект деблокируется, только когда завершается самый внешний оператор `lock` – или выполняется совпадающее количество операторов `Monitor.Exit`.

Вложенное блокирование удобно, когда один метод вызывает другой изнутри блокировки:

```
static readonly object _locker = new object();

static void Main()
{
  lock (_locker)
  {
    AnotherMethod();
    // Мы по-прежнему имеем блокировку, т.к. она является реентерабельной.
  }
}
static void AnotherMethod()
{
  lock (_locker) { Console.WriteLine ("Another method"); }
}
```

Поток может блокироваться только на первой (самой внешней) блокировке.

## Взаимоблокировки

Взаимоблокировка случается, когда каждый из двух потоков ожидает ресурс, удерживаемый другим потоком, так что ни один из них не может продолжить работу. Сказанное проще всего проиллюстрировать с помощью двух блокировок:

```
object locker1 = new object();
object locker2 = new object();

new Thread (() => {
                   lock (locker1)
                   {
                     Thread.Sleep (1000);
                     lock (locker2);  // Взаимоблокировка
                   }
                  }).Start();
lock (locker2)
{
  Thread.Sleep (1000);
  lock (locker1);  // Взаимоблокировка
}
```

Три и большее количество потоков могут породить более сложные цепочки взаимоблокировок.

В стандартной среде размещения система CLR не похожа на SQL Server; она не обнаруживает и не разрешает взаимоблокировки автоматически, принудительно прекращая работу одного из нарушителей. Взаимоблокировка потоков приводит к тому, что участвующие потоки блокируются на неопределенный срок, если только не был указан таймаут блокировки. (Тем не менее, под управлением хоста интеграции CLR с SQL Server взаимоблокировки *обнаруживаются* автоматически с генерацией перехватываемого исключения в одном из потоков.)

Взаимоблокировка является одной из самых сложных проблем многопоточности — особенно, когда есть множество взаимосвязанных объектов. По существу сложность кроется в том, что вы не можете с уверенностью сказать, какие блокировки получил *вызывающий поток*.

Таким образом, вы можете блокировать закрытое поле a внутри своего класса x, не зная, что вызывающий поток (или поток, обращающийся к вызывающему потоку) уже заблокировал поле b в классе y. Тем временем другой поток делает обратное, создавая взаимоблокировку. По иронии судьбы проблема усугубляется (хорошими) паттернами объектно-ориентированного проектирования, потому что паттерны подобного рода создают цепочки вызовов, которые не определены вплоть до стадии выполнения.

Хотя популярный совет блокировать объекты в согласованном порядке во избежание взаимоблокировок был полезен в начальном примере, он труден в применении к только что описанному сценарию. Лучшая стратегия заключается в том, чтобы проявлять осторожность при блокировании обращений к методам в объектах, которые могут иметь ссылки на ваш объект. Кроме того, следует подумать, действительно ли нужна блокировка обращений к методам в других классах (как будет показано в разделах, посвященных безопасности к потокам, это делается часто, но иногда доступны другие возможности). В большей степени полагаясь на высокоуровневые средства

синхронизации, такие как продолжения/комбинаторы задач, параллелизм данных и неизменяемые типы (рассматриваются далее в главе), потребность в блокировании можно снизить.

Существует альтернативный путь восприятия данной проблемы: когда вы обращаетесь к другому коду, удерживая блокировку, инкапсуляция такой блокировки незаметно *исчезает*. Это не ошибка в CLR или .NET Framework, а фундаментальное ограничение блокирования в целом. Проблемы блокирования решаются в рамках разнообразных исследовательских проектов, включая проект *Software Transactional Memory* (Программная транзакционная память).

Еще один сценарий взаимоблокировки возникает при вызове метода `Dispatcher.Invoke` (в приложении WPF) или `Control.Invoke` (в приложении Windows Forms) во время владения блокировкой. Если случится так, что пользовательский интерфейс выполняет другой метод, который ожидает ту же самую блокировку, то именно здесь и возникнет взаимоблокировка. Часто проблему можно устранить, просто вызывая метод `BeginInvoke` вместо `Invoke` (или положиться на асинхронные функции, которые делают это неявно, когда присутствует контекст синхронизации). В качестве альтернативы перед вызовом `Invoke` можно освободить свою блокировку, хотя прием не сработает, если блокировку отобрал *вызывающий поток*.

## Производительность

Блокировка выполняется быстро: можно ожидать, что получение и освобождение блокировки на современном компьютере займет менее 50 наносекунд при отсутствии соперничества за эту блокировку. В случае соперничества побочное переключение контекста смещает накладные расходы ближе к микросекундной области, хотя они могут оказаться еще больше перед тем, как действительно произойдет повторное планирование потока.

### Mutex

Класс `Mutex` похож на оператор `lock` языка C#, но он способен работать во множестве процессов. Другими словами, `Mutex` может иметь область действия на уровне *компьютера* и *приложения*. Получение и освобождение объекта `Mutex` требует около одной микросекунды при отсутствии соперничества, т.е. он примерно в 20 раз медленнее оператора `lock`.

В случае класса `Mutex` для блокирования вызывается его метод `WaitOne`, а для разблокирования – метод `ReleaseMutex`. Как и оператор `lock`, объект `Mutex` может быть освобожден из того же самого потока, в котором он был получен.

Если вы забудете вызвать метод `ReleaseMutex` и просто сделаете вызов `Close` или `Dispose`, то в любом другом потоке, ожидающем данный объект `Mutex`, сгенерируется исключение `AbandonedMutexException`.

Межпроцессный объект `Mutex` часто используется для обеспечения того, что в каждый момент времени может выполняться только один экземпляр программы. Ниже показано, как это делается.

```
class OneAtATimePlease
{
```

```
  static void Main()
  {
    // Назначение объекту Mutex имени делает его доступным
    // на уровне всего компьютера.
    // Используйте имя, являющееся уникальным для вашей компании и приложения
    // (например, включите в него URL компании).
    using (var mutex = new Mutex (true, "oreilly.com OneAtATimeDemo"))
    {
      // Ожидать несколько секунд, если возникло соперничество; в этом случае
      // другой экземпляр программы все еще находится в процессе завершения.
      if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))
      {
        Console.WriteLine ("Another instance of the app is running. Bye!");
                          // Выполняется другой экземпляр программы
        return;
      }
      try { RunProgram(); }
      finally { mutex.ReleaseMutex (); }
    }
  }
  static void RunProgram()
  {
    Console.WriteLine ("Running. Press Enter to exit");
                       // Программа выполняется; нажмите Enter для завершения
    Console.ReadLine();
  }
}
```

При выполнении под управлением терминальных служб (Terminal Services) объект `Mutex` уровня компьютера обычно виден только приложениям в том же сеансе терминального сервера. Чтобы сделать его видимым всем сеансам терминального сервера, добавьте к его имени префикс `Global\`.

# Блокирование и безопасность к потокам

Программа или метод является безопасным в отношении потоков, если обладает способностью корректно работать в любом многопоточном сценарии. Безопасность к потокам достигается главным образом за счет блокирования и уменьшения возможностей взаимодействия потоков.

Универсальные типы редко бывают безопасными к потокам в полном объеме по перечисленным ниже причинам.

- Затраты при разработке, необходимые для обеспечения полной безопасности к потокам, могут оказаться значительными, особенно если тип имеет множество полей (каждое поле потенциально открыто для взаимодействия в произвольном многопоточном контексте).
- Безопасность к потокам может повлечь за собой снижение производительности (частично зависящее от того, применяется ли тип во множестве потоков).
- Безопасный в отношении потоков тип не обязательно автоматически превращает использующую его программу в безопасную к потокам. Часто работа, связанная с построением программы, делает избыточными усилия по достижению безопасности к потокам самого типа.

Таким образом, безопасность к потокам обычно реализуется только там, где она нужна, с целью поддержки специфичного многопоточного сценария.

Однако есть несколько способов "схитрить" и заставить крупные и сложные классы безопасно выполняться в многопоточной среде. Один из них предусматривает принесение в жертву степени детализации за счет помещения больших разделов кода – даже кода доступа ко всему объекту – внутрь единственной монопольной блокировки, обеспечивая последовательный доступ на высоком уровне. На самом деле такая тактика жизненно важна, если необходимо применять небезопасный к потокам код третьей стороны (или большинство типов .NET Framework, если уж на то пошло) в многопоточном контексте. Уловка заключается просто в использовании одной и той же монопольной блокировки для защиты доступа ко всем свойствам, методам и полям небезопасного к потокам объекта. Такое решение хорошо работает, если все методы объекта выполняются быстро (в противном случае будет много блокирования).

Оставив в стороне примитивные типы, лишь очень немногие типы .NET Framework позволяют создавать экземпляры, которые безопасны к потокам за рамками простого параллельного доступа только для чтения. Ответственность за обеспечение безопасности к потокам, обычно посредством монопольных блокировок, возлагается на разработчика. (Исключением являются коллекции из пространства имен `System.Collections.Concurrent`, которые мы рассмотрим в главе 23.)

Еще один способ схитрить предполагает минимизацию взаимодействия потоков за счет сведения к минимуму разделяемых данных. Это великолепный подход, который неявно применяется в лишенных состояния серверах приложений среднего уровня и серверах веб-страниц. Поскольку множественные клиентские запросы могут поступать одновременно, серверные методы, к которым они обращаются, должны быть безопасными к потокам. Проектное решение, при котором состояние не запоминается (популярное по причинам масштабируемости), по существу ограничивает возможность взаимодействия, т.к. классы не сохраняют данные между запросами. Взаимодействие потоков затем ограничивается только статическими полями, которые могут создаваться для таких целей, как кеширование часто используемых данных в памяти и предоставление инфраструктурных служб вроде аутентификации и аудита.

Другое решение (в насыщенных клиентских приложениях) предусматривает запуск кода, который получает доступ к разделяемому состоянию в потоке пользовательского интерфейса. Как было показано в главе 14, асинхронные функции упрощают реализацию такого подхода.

Последний подход по обеспечению безопасности к потокам предполагает работу в режиме автоматического блокирования. Именно это будет делать инфраструктура .NET Framework, если создать подкласс `ContextBoundObject` и применить к нему атрибут `Synchronization`. Всякий раз, когда впоследствии вызывается метод или свойство такого объекта, блокировка уровня объекта получается автоматически для всего периода выполнения метода или свойства. Хотя подход подобного рода снижает объем усилий, затрачиваемых на безопасность к потокам, он привносит собственные проблемы: взаимоблокировки, которые иначе не возникали бы, вырождающийся параллелизм и незапланированную реентерабельность. По указанным причинам ручное блокирование обычно является наилучшим вариантом – во всяком случае, пока не станет доступным не настолько упрощенческий режим автоматического блокирования.

## Безопасность к потокам и типы .NET Framework

Блокирование может использоваться для преобразования небезопасного к потокам кода в код, безопасный в отношении потоков. Хорошим сценарием его применения следует считать инфраструктуру .NET Framework: почти все непримитивные типы в ней не являются безопасными к потокам (когда задачи выходят за рамки простого доступа только для чтения), но, тем не менее, они могут использоваться в многопоточном коде, если весь доступ к любому заданному объекту защищен посредством блокировки. Ниже приведен пример, в котором два потока одновременно добавляют элемент к одной и той же коллекции `List`, после чего выполняют перечисление этой коллекции:

```
class ThreadSafe
{
  static List <string> _list = new List <string>();

  static void Main()
  {
    new Thread (AddItem).Start();
    new Thread (AddItem).Start();
  }

  static void AddItem()
  {
    lock (_list) _list.Add ("Item " + _list.Count);

    string[] items;
    lock (_list) items = _list.ToArray();
    foreach (string s in items) Console.WriteLine (s);
  }
}
```

В данном случае мы блокируем сам объект `_list`. Если бы существовали два взаимосвязанных списка, то для применения блокировки мы должны были бы выбрать общий объект (на его место можно было бы назначить один из списков или, что еще лучше — использовать независимое поле).

Перечисление коллекций .NET также не является безопасным к потокам в том смысле, что если список изменяется во время перечисления, тогда генерируется исключение. В приведенном примере вместо блокирования на протяжении всего перечисления мы сначала копируем элементы в массив, что позволяет избежать удержания блокировки чрезмерно долго, если действия, предпринимаемые при перечислении, потенциально могут отнимать много времени. (Другое решение заключается в том, чтобы применить блокировку объекта чтения/записи, как объясняется в разделе "Блокировки объектов чтения/записи" далее в главе.)

### Блокирование безопасных к потокам объектов

Иногда блокировку также необходимо применять при доступе к объектам, безопасным в отношении потоков. В целях иллюстрации предположим, что класс `List` из .NET Framework на самом деле безопасен к потокам, и нужно добавить элемент в список:

```
if (!_list.Contains (newItem)) _list.Add (newItem);
```

Вне зависимости от того, безопасен список к потокам или нет, приведенный оператор таковым определенно не является! Весь этот оператор `if` должен быть помещен внутрь блокировки, чтобы предотвратить вытеснение в промежутке между проверкой наличия элемента в списке и добавлением нового элемента. Ту же самую блокировку затем нужно использовать везде, где список модифицируется. Например, следующий

оператор также требует помещения в идентичную блокировку, чтобы его не вытеснил предшествующий оператор:

```
_list.Clear();
```

Другими словами, нам пришлось бы применять блокировки точно так же, как мы поступали с классами небезопасных к потокам коллекций (делая гипотетическую безопасность к потокам класса `List` избыточной).

Применение блокировки к коду доступа в коллекцию может привести к чрезмерному блокированию в средах с высокой степенью параллелизма. Именно потому, начиная с .NET Framework 4.0, предлагаются безопасные к потокам версии очереди, стека и словаря, которые обсуждаются в главе 23.

### Статические члены

Помещение кода доступа к объекту внутрь специальной блокировки работает, только если все параллельные потоки осведомлены — и используют — данную блокировку. Это может быть не так, если объект имеет широкую область видимости. Худший случай касается статических членов в открытом типе. Например, представьте ситуацию, когда статическое свойство структуры `DateTime`, такое как `DateTime.Now`, не является безопасным к потокам, и два параллельных вызова могут дать в результате искаженный вывод либо исключение. Единственный способ устранить проблему с помощью внешнего блокирования может предусматривать блокировку самого типа — `lock(typeof(DateTime))` — перед вызовом `DateTime.Now`. Прием сработает, только если все программисты согласятся поступать подобным образом (что маловероятно). Более того, блокировка типа привносит собственные проблемы.

По указанной причине статические члены структуры `DateTime` были осмотрительно запрограммированы как безопасные к потокам. Такой шаблон применяется в .NET Framework повсеместно: *статические члены являются безопасными к потокам, а члены экземпляра – нет.* Следовать упомянутому шаблону также имеет смысл при написании типов для общественного потребления, поскольку он позволяет избежать создания неразрешимых проблем с безопасностью в отношении потоков. Другими словами, делая статические методы безопасными к потокам, вы программируете так, чтобы не *препятствовать* безопасности к потокам для потребителей данного типа.

Безопасность к потокам в статических методах придется кодировать явным образом: она не возникает автоматически только в силу того, что метод является статическим!

### Безопасность к потокам для доступа только по чтению

Превращение типов в безопасные к потокам для параллельного доступа только по чтению (там, где возможно) дает преимущество в том, что потребители могут избежать излишнего блокирования. Данный принцип соблюдают многие типы в .NET Framework: например, коллекции являются безопасными к потокам для параллельных объектов чтения.

Следовать такому принципу довольно просто: если вы документировали тип как безопасный к потокам для параллельного доступа только по чтению, то не производите запись в поля внутри методов, которые по ожиданиям потребителя должны допускать только чтение (или помещаете такой код внутрь блокировки). Например, реализация метода `ToArray` в коллекции может начинаться с уплотнения внутренней структуры

коллекции. Тем не менее, это сделало бы метод небезопасным к потокам для потребителей, которые ожидают, что он допускает только чтение.

Безопасность к потокам для доступа только по чтению является одной из причин, по которым перечислители отделены от классов, поддерживающих перечисление: два потока могут одновременно перечислять коллекцию, потому что каждый из них получает отдельный объект перечислителя.

Если документация по типу отсутствует, тогда имеет смысл проявлять осторожность в предположениях о том, что тот или иной метод по своей природе является предназначенным только для чтения. Хорошим примером может служить класс `Random`: при вызове метода `Random.Next` его внутренняя реализация требует обновления закрытых начальных значений. Следовательно, вы должны либо применять блокировку к коду, использующему класс `Random`, либо поддерживать отдельные его экземпляры для каждого потока.

## Безопасность к потокам в серверах приложений

Серверы приложений должны быть многопоточными, чтобы обрабатывать одновременные клиентские запросы. Приложения WCF, ASP.NET и Web Services многопоточны неявно; то же самое справедливо в отношении серверных приложений Remoting, которые имеют дело с сетевым каналом вроде TCP или HTTP. Это означает, что при написании кода на серверной стороне вы должны принимать во внимание безопасность к потокам, если есть хотя бы малейшая возможность взаимодействия между потоками, обрабатывающими клиентские запросы. К счастью, такая возможность возникает редко; типичный серверный класс либо не сохраняет состояние (поля отсутствуют), либо имеет модель активизации, которая создает отдельный его экземпляр для каждого клиента или каждого запроса. Взаимодействие обычно происходит только через статические поля, которые иногда применяются для кеширования в памяти частей базы данных с целью повышения производительности.

Например, предположим, что имеется метод `RetrieveUser`, выдающий запрос к базе данных:

```
// User - специальный класс с полями для хранения данных о пользователе
internal User RetrieveUser (int id) { ... }
```

Если метод `RetrieveUser` вызывается часто, тогда показатели производительности можно было бы улучшить, кешируя результаты в статическом объекте `Dictionary`. Ниже показано решение, учитывающее безопасность к потокам:

```
static class UserCache
{
  static Dictionary <int, User> _users = new Dictionary <int, User>();
  internal static User GetUser (int id)
  {
    User u = null;
    lock (_users)
      if (_users.TryGetValue (id, out u))
        return u;
    u = RetrieveUser (id);  // Метод для извлечения информации из базы данных
    lock (_users) _users [id] = u;
    return u;
  }
}
```

Для обеспечения безопасности в отношении потоков мы должны, как минимум, применить блокировку к чтению и обновлению словаря. В приведенном примере мы отдаем предпочтение практичному компромиссу между простотой и производительностью в блокировании. Наше проектное решение на самом деле создает очень небольшой потенциал для неэффективности: если два потока одновременно вызовут данный метод с одним и тем же ранее не извлеченным идентификатором `id`, то метод `RetrieveUser` будет вызван дважды – и словарь обновится лишний раз. Одиночное блокирование всего метода могло бы предотвратить такую ситуацию, но породить серьезную неэффективность: на протяжении вызова `RetrieveUser` блокировался бы целый кеш, и в это время другие потоки не могли бы извлекать информацию о *любых* пользователях.

## Неизменяемые объекты

Неизменяемым является такой объект, состояние которого не может быть модифицировано, ни внешне, ни внутренне. Поля в неизменяемом объекте обычно объявляются как предназначенные только для чтения и полностью инициализируются во время его конструирования.

Неизменяемость является признаком функционального программирования, где вместо *изменения* существующего объекта создается новый объект с отличающимися свойствами. Указанной парадигме следует язык LINQ. Неизменяемость также полезна в случае многопоточности – она позволяет избежать проблемы записываемого разделяемого состояния, устраняя (или сводя к минимуму) возможность записи.

Один из шаблонов предусматривает использование неизменяемых объектов для инкапсуляции группы связанных полей, чтобы снизить до минимума продолжительность действия блокировок. В качестве очень простого примера предположим, что имеются два следующих поля:

```
int _percentComplete;
string _statusMessage;
```

которые требуется читать/записывать атомарным образом. Вместо применения блокировки к этим полям мы можем определить неизменяемый класс, как показано ниже:

```
class ProgressStatus  // Представляет ход некоторого действия
{
  public readonly int PercentComplete;
  public readonly string StatusMessage;

  // Этот класс может иметь намного больше полей...

  public ProgressStatus (int percentComplete, string statusMessage)
  {
    PercentComplete = percentComplete;
    StatusMessage = statusMessage;
  }
}
```

Затем можно определить одиночное поле такого типа вместе с объектом блокировки:

```
readonly object _statusLocker = new object();
ProgressStatus _status;
```

Теперь значения типа ProgressStatus можно читать/записывать, не удерживая блокировку для чего-то большего, чем одиночное присваивание:

```
var status = new ProgressStatus (50, "Working on it");
// Здесь можно было бы выполнять присваивание многих других полей...
// ...
lock (_statusLocker) _status = status;  // Очень короткая блокировка
```

Чтобы прочитать объект, мы сначала получаем копию ссылки на него (внутри блокировки). Затем мы можем читать его значения без необходимости в удержании блокировки:

```
ProgressStatus status;
lock (_statusLocker) status = _status;  // И снова короткая блокировка
int pc = status.PercentComplete;
string msg = status.StatusMessage;
...
```

# Немонопольное блокирование

## Семафор

Семафор чем-то похож на ночной клуб: он обладает определенной вместительностью, за которой следит вышибала. Как только клуб переполнится, никто в него больше не сможет войти, и снаружи образуется очередь. Затем вместо каждого покинувшего клуб человека туда входит один человек из головы очереди. Конструктор требует минимум двух аргументов: количества свободных в текущий момент мест и общей вместимости ночного клуба.

Семафор с вместительностью, составляющей единицу, подобен Mutex или lock за исключением того, что семафор не имеет "владельца" — он *независим от потоков*. Любой поток способен вызывать метод Release объекта Semaphore, тогда как в случае Mutex и lock освободить блокировку может только поток, который ее получил.

Существуют две функционально похожие версии данного класса: Semaphore и SemaphoreSlim. Последняя версия была введена в .NET Framework 4.0 и оптимизирована для удовлетворения требованиям низкой задержки, которые предъявляются параллельным программированием. Она также полезна при традиционном многопоточном программировании, т.к. позволяет указывать признак отмены во время ожидания (см. раздел "Отмена" в главе 14) и открывает доступ к методу WaitAsync для асинхронного программирования. Тем не менее, SemaphoreSlim не может использоваться для сигнализирования между процессами.

Класс Semaphore требует около 1 микросекунды при вызове метода WaitOne или Release, а класс SemaphoreSlim — примерно одну десятую этого времени.

Семафоры могут быть удобны для ограничения параллелизма, предотвращая выполнение отдельной порции кода слишком большим количеством потоков. В следующем примере пять потоков пытаются войти в ночной клуб, который разрешает вход только трем потокам одновременно:

```
class TheClub  // Никаких списков дверей!
{
  static SemaphoreSlim _sem = new SemaphoreSlim (3); //Вместительность равна 3
```

```
  static void Main()
  {
    for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
  }
  static void Enter (object id)
  {
    Console.WriteLine (id + " wants to enter");
    _sem.Wait();
    Console.WriteLine (id + " is in!");       // Одновременно здесь
    Thread.Sleep (1000 * (int) id);           // могут находиться
    Console.WriteLine (id + " is leaving");   // только три потока.
    _sem.Release();
  }
}
```

```
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

Если объекту `Semaphore` назначено имя, тогда он может охватывать множество процессов тем же способом, что и `Mutex`.

## Блокировки объектов чтения/записи

Довольно часто экземпляры типа являются безопасными в отношении потоков для параллельных операций чтения, но не для параллельных обновлений (и не для параллельных операций чтения с обновлением). Это также может быть верным для ресурсов, подобных файлам. Хотя защита экземпляров таких типов посредством простой монопольной блокировки для всех режимов доступа обычно требует ухищрений, она может чрезмерно ограничить параллелизм, когда существует много операций чтения и только несколько операций обновления. Примером, когда такая ситуация может возникнуть, является сервер бизнес-приложений, где часто используемые данные кешируются в статических полях с целью их быстрого извлечения. Класс `ReaderWriterLockSlim` предназначен для обеспечения блокирования с максимальной доступностью именно в таких сценариях.

Класс `ReaderWriterLockSlim` появился в версии .NET Framework 3.5 и представляет собой замену более старого "тяжеловесного" класса `ReaderWriterLock`. Класс `ReaderWriterLock` обладает похожей функциональностью, но в несколько раз медленнее и содержит внутреннюю проектную ошибку в механизме, который отвечает за обработку повышений уровня блокировок.

Тем не менее, по сравнению с обычным оператором `lock` (`Monitor.Enter`/`Monitor.Exit`) класс `ReaderWriterLockSlim` все равно работает в два раза медленнее. Компромиссом является меньшая степень соперничества (когда производится много чтения и минимум записи).

С обоими классами связаны два базовых вида блокировок – блокировка чтения и блокировка записи:

- блокировка записи является универсально монопольной;
- блокировка чтения совместима с другими блокировками чтения.

Следовательно, поток, удерживающий блокировку записи, блокирует все другие потоки, которые пытаются получить блокировку чтения *или* записи (и наоборот). Но если потоки, удерживающие блокировку записи, отсутствуют, тогда любое количество потоков может параллельно получить блокировку чтения.

В классе `ReaderWriterLockSlim` определены методы для получения и освобождения блокировок чтения/записи:

```
public void EnterReadLock();
public void ExitReadLock();
public void EnterWriteLock();
public void ExitWriteLock();
```

Вдобавок есть версии `Try` всех методов `Enter`*XXX*, которые принимают аргументы тайм-аута в стиле метода `Monitor.TryEnter` (тайм-ауты могут происходить довольно часто, если ресурс подвержен серьезному соперничеству). Класс `ReaderWriterLock` предлагает аналогичные методы, именуемые `Acquire`*XXX* и `Release`*XXX*. Когда случается тайм-аут, вместо возвращения `false` они генерируют исключение `ApplicationException`.

В приведенной ниже программе демонстрируется применение класса `ReaderWriterLockSlim`. Три потока постоянно выполняют перечисление списка, в то время как два других потока ежесекундно добавляют в список случайное число. Блокировка чтения защищает потоки, читающие список, а блокировка записи – потоки, выполняющие запись в список.

```
class SlimDemo
{
  static ReaderWriterLockSlim _rw = new ReaderWriterLockSlim();
  static List<int> _items = new List<int>();
  static Random _rand = new Random();

  static void Main()
  {
    new Thread (Read).Start();
    new Thread (Read).Start();
    new Thread (Read).Start();

    new Thread (Write).Start ("A");
    new Thread (Write).Start ("B");
  }

  static void Read()
  {
    while (true)
    {
      _rw.EnterReadLock();
      foreach (int i in _items) Thread.Sleep (10);
      _rw.ExitReadLock();
    }
  }
```

```
  static void Write (object threadID)
  {
    while (true)
    {
      int newNumber = GetRandNum (100);
      _rw.EnterWriteLock();
      _items.Add (newNumber);
      _rw.ExitWriteLock();
      Console.WriteLine ("Thread " + threadID + " added " + newNumber);
      Thread.Sleep (100);
    }
  }

  static int GetRandNum (int max) { lock (_rand) return _rand.Next(max); }
}
```

В производственный код обычно будут добавляться блоки `try/finally`, гарантирующие освобождение блокировок в случае генерации исключения.

Вот результат:

```
Thread B added 61
Thread A added 83
Thread B added 55
Thread A added 33
...
```

Класс `ReaderWriterLockSlim` делает возможным действие `Read` с большей степенью параллелизма, чем простая блокировка. Это можно проиллюстрировать помещением следующей строки в начало цикла `while` внутри метода `Write`:

```
Console.WriteLine (_rw.CurrentReadCount + " concurrent readers");
```

Данная строка почти всегда будет сообщать о наличии трех параллельных читающих потоков (*б*ольшую часть своего времени методы `Read` тратят внутри циклов `foreach`). Помимо `CurrentReadCount` класс `ReaderWriterLockSlim` предлагает следующие свойства для слежения за блокировками:

```
public bool IsReadLockHeld             { get; }
public bool IsUpgradeableReadLockHeld  { get; }
public bool IsWriteLockHeld            { get; }

public int  WaitingReadCount           { get; }
public int  WaitingUpgradeCount        { get; }
public int  WaitingWriteCount          { get; }

public int  RecursiveReadCount         { get; }
public int  RecursiveUpgradeCount      { get; }
public int  RecursiveWriteCount        { get; }
```

### Блокировки с возможностью повышения уровня

Иногда в одиночной атомарной операции удобно поменять блокировку чтения на блокировку записи. Например, предположим, что вы хотите добавлять элемент в список, только если этот элемент в списке отсутствует. В идеальном случае желательно минимизировать время, затрачиваемое на удержание (монопольной) блокировки записи, а потому можно поступить так, как описано ниже.

1. Получить блокировку чтения.
2. Проверить, существует ли элемент в списке, и если существует, тогда освободить блокировку и произвести возврат.
3. Освободить блокировку чтения.
4. Получить блокировку записи.
5. Добавить элемент.

Проблема в том, что между шагом 3 и шагом 4 может проскользнуть другой поток и модифицировать список (например, добавив тот же самый элемент). Класс `ReaderWriterLockSlim` решает такую проблему через блокировку третьего вида, которая называется *блокировкой с возможностью повышения уровня*. Блокировка с возможностью повышения уровня похожа на блокировку чтения за исключением того, что позже она может быть повышена до уровня блокировки записи в атомарной операции. Вот как ее использовать.

1. Вызвать метод `EnterUpgradeableReadLock`.
2. Выполнить действия, связанные с чтением (например, проверить, существует ли элемент в списке).
3. Вызвать метод `EnterWriteLock` (что преобразует блокировку с возможностью повышения уровня в блокировку записи).
4. Выполнить действия, связанные с записью (например, добавить элемент в список).
5. Вызвать метод `ExitWriteLock` (что преобразует блокировку записи обратно в блокировку с возможностью повышения уровня).
6. Выполнить любые другие действия, связанные с чтением.
7. Вызвать метод `ExitUpgradeableReadLock`.

С точки зрения вызывающего кода все довольно похоже на вложенное или рекурсивное блокирование. Тем не менее, функционально на третьем шаге `ReaderWriterLockSlim` освобождает блокировку чтения и получает новую блокировку записи атомарным образом.

Между блокировками с возможностью повышения уровня и блокировками чтения имеется еще одно важное отличие. Несмотря на то что блокировка с возможностью повышения уровня способна сосуществовать с любым количеством блокировок *чтения*, в каждый момент времени может быть получена только одна блокировка с возможностью повышения уровня. Это предотвращает взаимоблокировки преобразований за счет сериализации соперничающих преобразований — почти как в случае блокировок обновлений в SQL Server:

| SQL Server | `ReaderWriterLockSlim` |
|---|---|
| Разделяемая блокировка | Блокировка чтения |
| Монопольная блокировка | Блокировка записи |
| Блокировка обновления | Блокировка с возможностью повышения уровня |

Мы можем продемонстрировать работу блокировки с возможностью повышения уровня, изменив метод `Write` из предыдущего примера так, чтобы он добавлял в список число, только если оно в нем отсутствует:

```
while (true)
{
  int newNumber = GetRandNum (100);
  _rw.EnterUpgradeableReadLock();
  if (!_items.Contains (newNumber))
  {
    _rw.EnterWriteLock();
    _items.Add (newNumber);
    _rw.ExitWriteLock();
    Console.WriteLine ("Thread " + threadID + " added " + newNumber);
  }
  _rw.ExitUpgradeableReadLock();
  Thread.Sleep (100);
}
```

Класс ReaderWriterLock также может выполнять преобразования блокировок, но ненадежно, потому что он не поддерживает концепцию блокировок с возможностью повышения уровня. Именно потому проектировщикам класса ReaderWriterLockSlim пришлось начинать полностью с нового класса.

## Рекурсия блокировок

Обычно вложенное или рекурсивное блокирование с участием класса ReaderWriterLockSlim запрещено. Таким образом, следующий код сгенерирует исключение:

```
var rw = new ReaderWriterLockSlim();
rw.EnterReadLock();
rw.EnterReadLock();
rw.ExitReadLock();
rw.ExitReadLock();
```

Однако он выполнится без ошибок, если объект ReaderWriterLockSlim конструируется так:

```
var rw = new ReaderWriterLockSlim (LockRecursionPolicy.SupportsRecursion);
```

Это гарантирует, что рекурсивное блокирование может произойти, только если оно запланировано. Рекурсивное блокирование может создать нежелательную сложность, т.к. появляется возможность получить более одного вида блокировок:

```
rw.EnterWriteLock();
rw.EnterReadLock();
Console.WriteLine (rw.IsReadLockHeld);   // True
Console.WriteLine (rw.IsWriteLockHeld);  // True
rw.ExitReadLock();
rw.ExitWriteLock();
```

Базовое правило гласит, что после получения блокировки последующие рекурсивные блокировки могут быть меньше, но не больше следующей шкалы:

*Блокировка чтения → Блокировка с возможностью повышения уровня → Блокировка записи*

Тем не менее, запрос на повышение блокировки с возможностью повышения уровня до блокировки записи законен всегда.

# Сигнализирование с помощью дескрипторов ожидания событий

Простейшая разновидность сигнализирующих конструкций называется *дескрипторами ожидания событий* (они никак не связаны с событиями C#). Дескрипторы ожидания событий поступают в трех формах: `AutoResetEvent`, `ManualResetEvent` (`ManualResetEventSlim`) и `CountdownEvent`. Первые две формы основаны на общем классе `EventWaitHandle`, от которого происходит вся их функциональность.

## AutoResetEvent

Класс `AutoResetEvent` похож на турникет: вставка билета позволяет пройти в точности одному человеку. Наличие слова `Auto` в имени класса отражает тот факт, что открытый турникет автоматически закрывается или "сбрасывается" после того, как кто-то через него прошел. Поток ожидает, или блокируется, на турникете вызовом метода `WaitOne` (ожидает до тех пор, пока этот "один" турникет не откроется), а билет вставляется вызовом метода `Set`. Если метод `WaitOne` вызван несколькими потоками, тогда перед турникетом выстраивается очередь[2]. Билет может поступать из любого потока; другими словами, любой (неблокированный) поток с доступом к объекту `AutoResetEvent` может вызвать на нем метод `Set` для освобождения одного заблокированного потока.

Создать объект `AutoResetEvent` можно двумя способами. Первый из них – применение конструктора:

```
var auto = new AutoResetEvent (false);
```

(Передача конструктору значения `true` эквивалентна немедленному вызову метода `Set` на результирующем объекте.) Второй способ создания объекта `AutoResetEvent` выглядит следующим образом:

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

В приведенном далее примере запускается поток, работа которого заключается в том, чтобы просто ожидать, пока он не будет сигнализирован другим потоком (рис. 22.1):

```
class BasicWaitHandle
{
  static EventWaitHandle _waitHandle = new AutoResetEvent (false);
  static void Main()
  {
    new Thread (Waiter).Start();
    Thread.Sleep (1000);       // Пауза в течение секунды...
    _waitHandle.Set();         // Пробудить Waiter.
  }
  static void Waiter()
  {
    Console.WriteLine ("Waiting...");
    _waitHandle.WaitOne();     // Ожидание уведомления
    Console.WriteLine ("Notified");
  }
}
// Вывод:
Waiting... (пауза) Notified.
```

[2] Как и в случае блокировок, равноправие в такой очереди временами может нарушаться из-за нюансов поведения операционной системы.

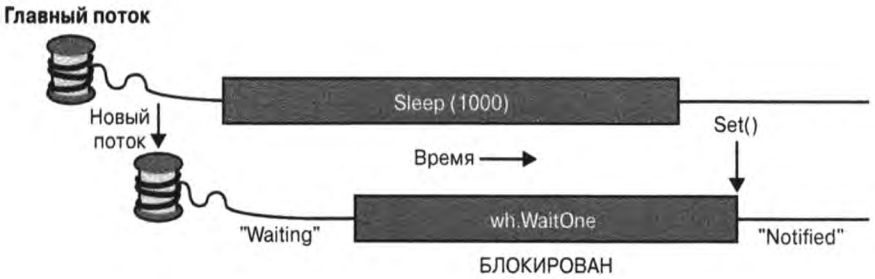



***Рис. 22.1.*** *Сигнализирование с помощью* `EventWaitHandle`

Если метод `Set` вызван, когда нет ни одного ожидающего потока, то дескриптор остается открытым до тех пор, пока он не дождется вызова метода `WaitOne` каким-либо потоком. Такое поведение помогает избежать состязаний между потоком, направляющимся к турникету, и потоком, вставляющим билет. Тем не менее, неоднократный вызов `Set` на турникете, перед которым никто не ожидает, не позволяет пройти целой компании, когда она соберется: пройти будет разрешено только следующему человеку, а дополнительные билеты растратятся впустую.

Вызов метода `Reset` на объекте `AutoResetEvent` закрывает турникет (если он был открыт) без ожидания или блокирования.

Метод `WaitOne` принимает дополнительный параметр тайм-аута, возвращая `false`, если ожидание закончилось по тайм-ауту, а не из-за получения сигнала.

Вызов метода `WaitOne` с тайм-аутом, равным `0`, проверяет, является ли дескриптор ожидания "открытым", не блокируя вызывающий поток. Однако помните, что такое действие сбросит объект `AutoResetEvent`, если он открыт.

### Освобождение дескрипторов ожидания

По завершении работы с дескриптором ожидания можно вызвать его метод `Close`, чтобы освободить ресурс операционной системы. В качестве альтернативы можно просто удалить все ссылки на дескриптор ожидания и позволить сборщику мусора сделать всю работу в какой-то момент позже (дескрипторы ожидания реализуют шаблон освобождения, в соответствии с которым финализатор вызывает метод `Close`). Это один из немногих сценариев, в которых вполне приемлемо полагаться на такой запасной вариант, потому что с дескрипторами ожидания связаны легковесные накладные расходы ОС.

Дескрипторы ожидания освобождаются автоматически при выгрузке домена приложения.

## Двунаправленное сигнализирование

Давайте предположим, что главный поток должен сигнализировать рабочий поток три раза в какой-то строке. Если главный поток просто вызовет метод `Set` на дескрипторе ожидания несколько раз в быстрой последовательности, тогда второй или третий сигнал может потеряться, т.к. рабочему потоку необходимо время на обработку каждого сигнала.

Решение для главного потока предусматривает ожидание перед выдачей сигнала до тех пор, пока рабочий поток не будет готов, что можно сделать посредством еще одного объекта AutoResetEvent:

```
class TwoWaySignaling
{
  static EventWaitHandle _ready = new AutoResetEvent (false);
  static EventWaitHandle _go = new AutoResetEvent (false);
  static readonly object _locker = new object();
  static string _message;
  static void Main()
  {
    new Thread (Work).Start();
    _ready.WaitOne();  // Сначала ожидать готовности рабочего потока
    lock (_locker) _message = "ooo";
    _go.Set();         // Сообщить рабочему потоку о начале продвижения
    _ready.WaitOne();
    lock (_locker) _message = "aah"; // Предоставить рабочему потоку
                                     // другое сообщение
    _go.Set();
    _ready.WaitOne();
    lock (_locker) _message = null;  // Сигнализировать рабочий поток
                                     // о завершении
    _go.Set();
  }
  static void Work()
  {
    while (true)
    {
      _ready.Set();   // Указать на готовность
      _go.WaitOne();  // Ожидать поступления сигнала...
      lock (_locker)
      {
        if (_message == null) return;  // Аккуратно завершить
        Console.WriteLine (_message);
      }
    }
  }
}
// Вывод:
ooo
aah
```

Диаграмма процесса представлена на рис. 22.2.

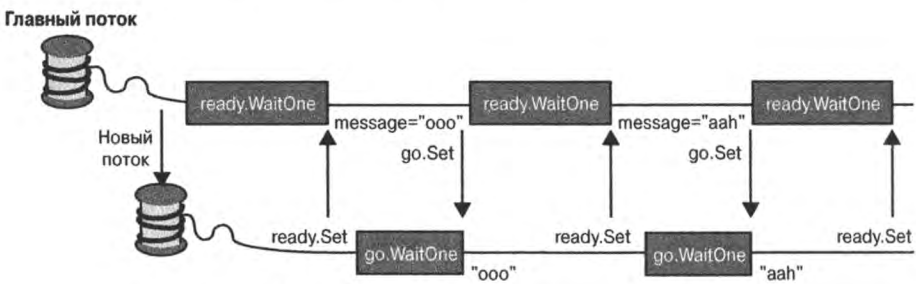



*Рис. 22.2. Двунаправленное сигнализирование*

Здесь сообщение null используется для указания на то, что рабочий поток должен завершиться. Для потоков, которые выполняются бесконечно, очень важно иметь стратегию завершения!

## ManualResetEvent

Как было описано в главе 14, объект ManualResetEvent функционирует подобно простым воротам. Вызов метода Set открывает ворота, позволяя *любому* количеству потоков вызывать WaitOne, чтобы получить разрешение пройти. Вызов метода Reset закрывает ворота. Потоки, которые вызывают WaitOne на закрытых воротах, блокируются; когда ворота откроются в следующий раз, все эти потоки будут одновременно освобождены. Помимо упомянутых отличий объект ManualResetEvent функционирует подобно объекту AutoResetEvent.

Как и AutoResetEvent, объект ManualResetEvent можно конструировать двумя путями:

```
var manual1 = new ManualResetEvent (false);
var manual2 = new EventWaitHandle (false, EventResetMode.ManualReset);
```

Начиная с .NET Framework 4.0, доступна еще одна версия класса ManualResetEvent под названием ManualResetEventSlim. Она оптимизирована под краткие периоды ожидания — с возможностью выбора зацикливания для установленного количества итераций. Класс ManualResetEventSlim также имеет более эффективную управляемую реализацию и позволяет методу Wait быть отмененным через CancellationToken. Тем не менее, данный класс не может применяться для межпроцессного сигнализирования. Класс ManualResetEventSlim не является подклассом WaitHandle; однако он открывает доступ к свойству WaitHandle, которое возвращает основанный на WaitHandle объект (с профилем производительности традиционного дескриптора ожидания).

### Сигнализирующие конструкции и производительность

Ожидание или сигнализирование AutoResetEvent либо ManualResetEvent занимают около одной микросекунды (при отсутствии блокирования).

Классы ManualResetEventSlim и CountdownEvent могут быть до 50 раз быстрее в сценариях с кратким ожиданием, поскольку они не зависят от операционной системы и благоразумно используют конструкции зацикливания.

Тем не менее, в большинстве сценариев накладные расходы, связанные с самими сигнализирующими классами, не создают узких мест, поэтому редко принимаются во внимание.

Класс ManualResetEvent удобен в предоставлении одному потоку возможности разблокировать множество других потоков. Обратный сценарий покрывается классом CountdownEvent.

## CountdownEvent

Класс CountdownEvent позволяет организовать ожидание на нескольких потоках. Он был введен в версии .NET Framework 4.0 и обладает эффективной полностью уп-

равляемой реализацией. Для применения данного класса создайте его экземпляр с нужным количеством потоков или "счетчиков", на которых необходимо ожидать:

```
var countdown = new CountdownEvent (3);  // Инициализировать со "счетчиком",
                                         // равным 3
```

Вызов метода Signal декрементирует счетчик; вызов метода Wait приводит к блокированию до тех пор, пока счетчик не станет равным нулю. Например:

```
static CountdownEvent _countdown = new CountdownEvent (3);

static void Main()
{
  new Thread (SaySomething).Start ("I am thread 1");
  new Thread (SaySomething).Start ("I am thread 2");
  new Thread (SaySomething).Start ("I am thread 3");
  _countdown.Wait();      // Блокируется до тех пор, пока Signal
                          // не будет вызван 3 раза
  Console.WriteLine ("All threads have finished speaking!");
}

static void SaySomething (object thing)
{
  Thread.Sleep (1000);
  Console.WriteLine (thing);
  _countdown.Signal();
}
```

Задачи, при решении которых эффективно использовать класс CountdownEvent, иногда могут решаться более просто с применением конструкций *структурированного параллелизма*, которые будут рассматриваться в главе 23 (PLINQ и класс Parallel).

Повторно инкрементировать счетчик CountdownEvent можно вызовом метода AddCount. Однако если он уже достиг нуля, то такой вызов приведет к генерации исключения: "отменить сигнал" CountdownEvent вызовом метода AddCount нельзя. Чтобы избежать возможности возникновения исключения, можно вызвать метод TryAddCount, который возвращает false, если счетчик достиг нуля.

Для отмены сигнала CountdownEvent необходимо вызвать метод Reset: он и отменит сигнал, и сбросит счетчик в исходное значение.

Подобно ManualResetEventSlim класс CountdownEvent открывает свойство WaitHandle для сценариев, в которых какой-то другой класс или метод ожидает объект, основанный на WaitHandle.

## Создание межпроцессного объекта EventWaitHandle

Конструктор EventWaitHandle позволяет "именовать" создаваемый объект EventWaitHandle, что дает ему возможность действовать в нескольких процессах. Имя — это просто строка, которая может иметь любое значение, не конфликтующее с именем какого-то другого объекта. Если указанное имя уже используется на данном компьютере, то вы получите ссылку на связанный с ним объект EventWaitHandle; в противном случае операционная система создаст новый объект. Ниже приведен пример:

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.AutoReset,
                                          "MyCompany.MyApp.SomeName");
```

Если данный код запускают два приложения, то они получат возможность сигнализировать друг друга: дескриптор ожидания будет работать для всех потоков в обоих процессах.

## Дескрипторы ожидания и продолжение

Вместо того чтобы ждать на дескрипторе ожидания (и тем самым блокировать поток), к нему можно присоединить "продолжение", вызвав метод `ThreadPool.RegisterWaitForSingleObject`, который принимает делегат, выполняющийся, когда дескриптор ожидания сигнализирован:

```
static ManualResetEvent _starter = new ManualResetEvent (false);

public static void Main()
{
  RegisteredWaitHandle reg = ThreadPool.RegisterWaitForSingleObject
   (_starter, Go, "Some Data", -1, true);
  Thread.Sleep (5000);
  Console.WriteLine ("Signaling worker...");
  _starter.Set();
  Console.ReadLine();
  reg.Unregister (_starter);  // Произвести очистку, когда все сделано.
}

public static void Go (object data, bool timedOut)
{
  Console.WriteLine ("Started - " + data);
  // Выполнить задачу...
}

// Вывод:
(пятисекундная задержка)
Signaling worker...
Started - Some Data
```

Когда дескриптор ожидания сигнализируется (либо истекает время тайм-аута), делегат запускается в потоке из пула. Затем понадобится вызвать метод `Unregister` для освобождения неуправляемого дескриптора обратного вызова.

В дополнение к дескриптору ожидания и делегату метод `RegisterWaitForSingleObject` принимает объект "черного ящика", который передается методу делегата (подобно `ParameterizedThreadStart`), а также длительность тайм-аута в миллисекундах (`-1` означает отсутствие тайм-аута) и булевский флаг, указывающий, является запрос одноразовым или повторяющимся.

## Преобразование дескрипторов ожидания в задачи

Работать с методом `ThreadPool.RegisterWaitForSingleObject` на практике затруднительно, потому что обычно нужно обращаться к методу `Unregister` из самого обратного вызова – до того, как признак регистрации станет доступным. Таким образом, имеет смысл написать расширяющий метод, который преобразует дескриптор ожидания в объект `Task`, допускающий применение к нему `await`:

```
public static Task<bool> ToTask (this WaitHandle waitHandle,
                                 int timeout = -1)
{
  var tcs = new TaskCompletionSource<bool>();
  RegisteredWaitHandle token = null;
```

```
  var tokenReady = new ManualResetEventSlim();
  token = ThreadPool.RegisterWaitForSingleObject (
    waitHandle,
    (state, timedOut) =>
    {
      tokenReady.Wait();
      tokenReady.Dispose();
      token.Unregister (waitHandle);
      tcs.SetResult (!timedOut);
    },
    null,
    timeout,
    true);
  tokenReady.Set();
  return tcs.Task;
}
```

Это позволяет присоединить продолжение к дескриптору ожидания:

```
myWaitHandle.ToTask().ContinueWith (...)
```

или применить к нему `await`:

```
await myWaitHandle.ToTask();
```

с необязательным тайм-аутом:

```
if (!await (myWaitHandle.ToTask (5000)))
  Console.WriteLine ("Timed out");
```

Обратите внимание, что в реализации `ToTask` мы использовали другой дескриптор ожидания (объект `ManualResetEventSlim`), чтобы избежать возникновения условий для состязаний, когда обратный вызов выполняется до присваивания переменной `token` признака регистрации.

## WaitAny, WaitAll и SignalAndWait

В дополнение к методам `Set`, `WaitOne` и `Reset` в классе `WaitHandle` определены статические методы, предназначенные для решения более сложных задач синхронизации. Методы `WaitAny`, `WaitAll` и `SignalAndWait` выполняют операции сигнализирования и ожидания на множестве дескрипторов. Дескрипторы ожидания могут быть разных типов (в том числе `Mutex` и `Semphore`, поскольку они также являются производными от абстрактного класса `WaitHandle`). Классы `ManualResetEventSlim` и `CountdownEvent` также могут принимать участие в указанных методах через свои свойства `WaitHandle`.

Методы `WaitAll` и `SignalAndWait` имеют странную связь с унаследованной архитектурой COM: они требуют, чтобы вызывающий поток находился в многопоточном апартаменте — модель, меньше всего подходящая для взаимодействия. Например, в таком режиме главный поток приложения WPF или Windows Forms не может взаимодействовать с буфером обмена. Вскоре мы обсудим доступные альтернативы.

Метод `WaitHandle.WaitAny` ожидает любой дескриптор ожидания из массива таких дескрипторов; метод `WaitHandle.WaitAll` ожидает все указанные дескрипторы атомарным образом. Это означает, что в случае ожидания двух объектов `AutoResetEvent`:

- метод WaitAny никогда не закончится "защелкиванием" обоих событий;
- метод WaitAll никогда не закончится "защелкиванием" только одного события.

Метод SignalAndWait вызывает Set на WaitHandle и затем WaitOne на другом WaitHandle. После сигнализирования первого дескриптора произойдет переход в начало очереди в ожидании второго дескриптора, что помогает ему двигаться вперед (хотя операция не является по-настоящему атомарной). Можете думать об этом методе, как о "подменяющем" один сигнал другим, и применять его на паре объектов EventWaitHandle для настройки двух потоков на рандеву, или "встречу", в одной и той же точке во времени. Такой трюк будет предпринимать либо AutoResetEvent, либо ManualResetEvent. Первый поток выполняет следующий вызов:

```
WaitHandle.SignalAndWait (wh1, wh2);
```

тогда как второй поток делает противоположное:

```
WaitHandle.SignalAndWait (wh2, wh1);
```

### Альтернативы методам WaitAll и SignalAndWait

Методы WaitAll и SignalAndWait не будут запускаться в однопоточном апартаменте. К счастью, существуют альтернативы. В случае SignalAndWait редко когда требуется его семантика перехода в начало очереди: скажем, в примере с рандеву было бы допустимо просто вызвать Set на первом дескрипторе ожидания и затем WaitOne на втором, если дескрипторы ожидания использовались исключительно для этого рандеву. В следующем разделе мы рассмотрим еще один вариант реализации рандеву потоков.

В случае методов WaitAny и WaitAll, если атомарность не нужна, то код, написанный в предыдущем разделе, можно применить для преобразования дескрипторов ожидания в задачи, после чего использовать методы Task.WhenAny и Task.WhenAll (см. главу 14).

Когда атомарность необходима, можно принять низкоуровневый подход к сигнализированию и самостоятельно написать логику с применением методов Wait и Pulse класса Monitor. Методы Wait и Pulse детально описаны на нашем веб-сайте по адресу http://albahari.com/threading/.

## Класс Barrier

Класс Barrier реализует *барьер выполнения потоков*, позволяя множеству потоков организовать рандеву в какой-то момент времени. Класс Barrier отличается высокой скоростью и эффективностью, и он построен на основе Wait, Pulse и спин-блокировок.

Для использования класса Barrier потребуется выполнить следующие действия.

1. Создать его экземпляр, указав количество потоков, которые должны принять участие в рандеву (позже их число можно изменить, вызывая методы AddParticipants и RemoveParticipants).
2. Заставить каждый поток вызвать метод SignalAndWait, когда он желает участвовать в рандеву.

Создание экземпляра Barrier со значением 3 приводит к блокированию вызова SignalAndWait до тех пор, пока данный метод не будет вызван три раза. Затем все начинается заново: вызов SignalAndWait снова блокируется, пока таких вызовов не станет три. Это сохраняет каждый поток "в синхронизме" с любым другим потоком.

В приведенном далее примере каждый из трех потоков выводит числа от 0 до 4, оставаясь в синхронизме с другими потоками:

```
static Barrier _barrier = new Barrier (3);

static void Main()
{
  new Thread (Speak).Start();
  new Thread (Speak).Start();
  new Thread (Speak).Start();
}

static void Speak()
{
  for (int i = 0; i < 5; i++)
  {
    Console.Write (i + " ");
    _barrier.SignalAndWait();
  }
}

ВЫВОД:  0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

Действительно полезная характеристика `Barrier` связана с возможностью указывать во время создания экземпляра также *действие, выполняемое после каждой фазы*. Такое действие представлено в виде делегата, который запускается после того, как метод `SignalAndWait` будет вызван *n* раз, но *перед* тем, как потоки деблокируются (как показано в затененной области на рис. 22.3). Если в рассматриваемом примере создать барьер следующим образом:

```
static Barrier _barrier = new Barrier (3, barrier => Console.WriteLine());
```

то вывод будет выглядеть так:

```
0 0 0
1 1 1
2 2 2
3 3 3
4 4 4
```

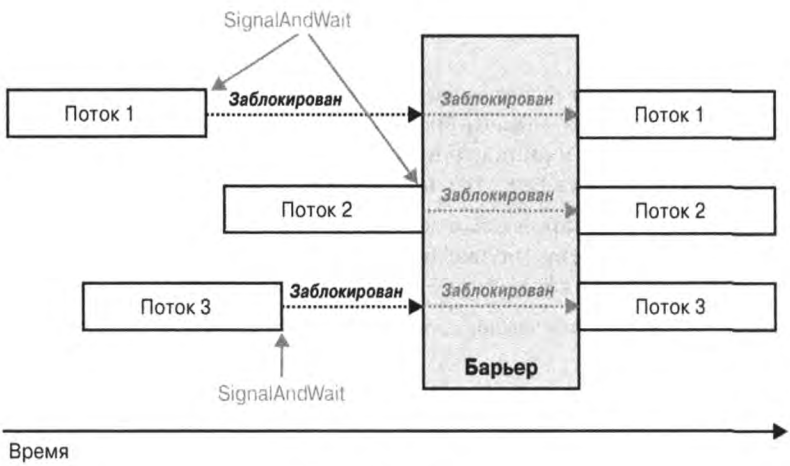



*Рис. 22.3. Барьер*

Действие, выполняемое после каждой фазы, может быть удобно для объединения данных из каждого рабочего потока. Беспокоиться о вытеснении не придется, потому что во время выполнения данного действия все рабочие потоки заблокированы.

# Ленивая инициализация

Частой проблемой в области многопоточности является определение способа ленивой инициализации разделяемого поля в манере, безопасной к потокам. Такая потребность возникает при наличии поля, которое относится к типу, затратному в плане конструирования:

```
class Foo
{
  public readonly Expensive Expensive = new Expensive();
  ...
}
class Expensive { /*Предположим, что это является затратным в конструировании*/ }
```

Проблема с показанным кодом заключается в том, что создание экземпляра `Foo` оказывает влияние на производительность из-за создания экземпляра класса `Expensive`, причем независимо от того, будет позже осуществляться доступ к полю `Expensive` или нет. Очевидное решение предусматривает конструирование экземпляра *по требованию*:

```
class Foo
{
  Expensive _expensive;
  public Expensive Expensive  // Ленивое создание экземпляра Expensive
  {
    get
    {
      if (_expensive == null) _expensive = new Expensive();
      return _expensive;
    }
  }
  ...
}
```

Здесь возникает вопрос: является ли такой код безопасным в отношении потоков? Оставив в стороне тот факт, что доступ к `_expensive` производится за пределами блокировки без барьера памяти, давайте подумаем, что произойдет, если два потока обратятся к данному свойству одновременно. Они оба могут дать `true` в условии оператора `if`, и каждый поток в конечном итоге получит *другой* экземпляр `Expensive`. Поскольку это может привести к возникновению тонких ошибок, в общем можно было бы сказать, что код не является безопасным к потокам.

Упомянутая проблема решается применением блокировки к коду проверки и инициализации объекта:

```
Expensive _expensive;
readonly object _expenseLock = new object();

public Expensive Expensive
{
  get
  {
```

```
    lock (_expenseLock)
    {
      if (_expensive == null) _expensive = new Expensive();
      return _expensive;
    }
  }
}
```

## Lazy<T>

Начиная с версии .NET Framework 4.0, стал доступным класс `Lazy<T>`, который помогает обеспечивать ленивую инициализацию. В случае создания его экземпляра с аргументом `true` он реализует только что описанный шаблон инициализации, безопасной в отношении потоков.

Класс `Lazy<T>` на самом деле реализует микро-оптимизированную версию этого шаблона, которая называется *блокированием с двойным контролем*. Блокирование с двойным контролем выполняет дополнительное временное (volatile) чтение, чтобы избежать затрат на получение блокировки, если объект уже инициализирован.

Для использования `Lazy<T>` создайте его экземпляр с делегатом фабрики значений, который сообщает, каким образом инициализировать новое значение, и аргументом `true`. Затем получайте доступ к его значению через свойство `Value`:

```
Lazy<Expensive> _expensive = new Lazy<Expensive>
  (() => new Expensive(), true);

public Expensive Expensive { get { return _expensive.Value; } }
```

Если конструктору класса `Lazy<T>` передать `false`, тогда он реализует шаблон ленивой инициализации, небезопасной к потокам, который был описан в начале настоящего раздела — это имеет смысл, когда класс `Lazy<T>` необходимо применять в однопоточном контексте.

## LazyInitializer

`LazyInitializer` — статический класс, который работает в точности как `Lazy<T>` за исключением перечисленных ниже моментов.

- Его функциональность открыта через статический метод, который оперирует прямо на поле вашего типа, что позволяет избежать дополнительного уровня косвенности, улучшая производительность в ситуациях, когда нужна высшая степень оптимизации.
- Он предлагает другой режим инициализации, при котором множество потоков могут состязаться за инициализацию.

Чтобы использовать класс `LazyInitializer`, перед доступом к полю необходимо вызвать его метод `EnsureInitialized`, передав ему ссылку на поле и фабричный делегат:

```
Expensive _expensive;
public Expensive Expensive
{
  get  // Реализовать блокирование с двойным контролем
  {
```

```
    LazyInitializer.EnsureInitialized (ref _expensive,
                                       () => new Expensive());
    return _expensive;
  }
}
```

Можно также передать еще один аргумент, чтобы запросить *состязание* за инициализацию конкурирующих потоков. Это звучит подобно исходному небезопасному к потокам примеру, исключая то, что первый пришедший к финишу поток всегда выигрывает – и потому в конечном итоге остается только один экземпляр. Преимущество такого приема связано с тем, что он даже быстрее (на многоядерных процессорах), чем блокирование с двойным контролем. Причина в том, что он может быть реализован полностью без блокировок с применением расширенных технологий, которые описаны в разделах "Nonblocking Synchronization" ("Неблокирующая синхронизация") и "Lazy Initialization" ("Ленивая инициализация") на нашем веб-сайте по адресу `http://albahari.com/threading/`. Это предельная (и редко востребованная) степень оптимизации, за которую придется заплатить определенную цену, как описано ниже.

- Такой подход будет медленнее, когда потоков, состязающихся за инициализацию, оказывается больше, чем ядер процессора.
- Потенциально он приводит к непроизводительным расходам ресурсов центрального процессора на выполнение избыточной инициализации.
- Логика инициализации обязана быть безопасной к потокам (в рассмотренном выше примере она может стать небезопасной к потокам, если конструктор `Expensive` будет производить запись в статические поля).
- Если инициализатор создает объект, требующий освобождения, то ставший "ненужным" такой объект не сможет быть освобожден без написания дополнительной логики.

# Локальное хранилище потока

Большая часть главы сосредоточена на конструкциях синхронизации и проблемах, возникающих из-за наличия у потоков возможности параллельного доступа к одним и тем же данным. Однако иногда данные должны храниться изолированно, гарантируя тем самым, что каждый поток имеет отдельную их копию. Именно этого позволяют добиться локальные переменные, но они пригодны только для переходных данных.

Решением является *локальное хранилище потока*. Здесь может возникнуть затруднение с пониманием требования: данные, которые вы хотели бы сохранить изолированными в потоке, как правило, являются переходными по своей природе. Основное использование локального хранилища касается хранения "внешних" данных, с помощью которых осуществляется поддержка инфраструктуры пути выполнения, такой как обмен сообщениями, транзакция и маркеры безопасности. Передача таких данных в параметрах методов является чрезвычайно неудобным и чуждым приемом для всех методов кроме тех, что написаны лично вами. С другой стороны, хранение такой информации в обычных статических полях означает ее разделение между всеми потоками.

Локальное хранилище потока может также быть полезным при оптимизации параллельного кода. Оно позволяет каждому потоку иметь монопольный доступ к собственной версии объекта, небезопасного к потокам, без необходимости в блокировке — и без потребности в воссоздании этого объекта между вызовами методов.

Тем не менее, локальное хранилище потока не очень хорошо сочетается с асинхронным кодом, потому что продолжение может выполняться в потоке, который отличается от предыдущего.

Существуют три способа реализации локального хранилища потока.

## [ThreadStatic]

Простейший подход к реализации локального хранилища потока предусматривает пометку статического поля с помощью атрибута `[ThreadStatic]`:

```
[ThreadStatic] static int _x;
```

После этого каждый поток будет видеть отдельную копию `_x`.

К сожалению, атрибут `[ThreadStatic]` не работает с полями экземпляра (он просто ничего не делает), а также не сочетается нормально с инициализаторами полей — в функционирующем потоке они выполняются только один раз, когда запускается статический конструктор. Если необходимо работать с полями экземпляра или начать с нестандартного значения, то более подходящим вариантом является `ThreadLocal<T>`.

## ThreadLocal<T>

Класс `ThreadLocal<T>` появился в версии .NET Framework 4.0. Он предоставляет локальное хранилище потока как для статических полей, так и для полей экземпляра — и вдобавок позволяет указывать стандартные значения.

Вот как создать объект `ThreadLocal<int>` со стандартным значением 3 для каждого потока:

```
static ThreadLocal<int> _x = new ThreadLocal<int> (() => 3);
```

Далее для получения или установки значения, локального для потока, применяется свойство `Value` объекта `_x`. Дополнительным преимуществом использования `ThreadLocal` является ленивая оценка значений: фабричная функция оценивается только при первом ее вызове (для каждого потока).

### ThreadLocal<T> и поля экземпляра

Класс `ThreadLocal<T>` также удобен при работе с полями экземпляра и захваченными локальными переменными. Например, рассмотрим задачу генерации случайных чисел в многопоточной среде. Класс `Random` не является безопасным в отношении потоков, поэтому мы должны либо применять блокировку вокруг кода, использующего `Random` (ограничивая степень параллелизма), либо генерировать отдельный объект `Random` для каждого потока. Класс `ThreadLocal<T>` делает второй подход простым:

```
var localRandom = new ThreadLocal<Random>(() => new Random());
Console.WriteLine (localRandom.Value.Next());
```

Указанная фабричная функция, создающая объект `Random`, несколько упрощена, т.к. конструктор без параметров класса `Random` при выборе начального значения для

генерации случайных чисел полагается на системные часы. Начальные значения могут оказаться одинаковыми для двух объектов Random, созданных внутри приблизительно 10-миллисекундного промежутка времени. Ниже продемонстрирован один из способов решения проблемы:

```
var localRandom = new ThreadLocal<Random>
  ( () => new Random (Guid.NewGuid().GetHashCode()) );
```

Мы будем использовать такой прием в следующей главе (в примере параллельной программы проверки орфографии внутри раздела "PLINQ").

## GetData и SetData

Третий подход предполагает применение двух методов класса Thread: GetData и SetData. Они сохраняют данные в "ячейках", специфичных для потока. Метод Thread.GetData выполняет чтение из изолированного хранилища данных потока, а метод Thread.SetData осуществляет запись в него. Оба метода требуют объекта LocalDataStoreSlot для идентификации ячейки. Одна и та же ячейка может использоваться во всех потоках, но они по-прежнему будут получать отдельные значения. Ниже приведен пример:

```
class Test
{
  // Один и тот же объект LocalDataStoreSlot может
  // использоваться во всех потоках.
  LocalDataStoreSlot _secSlot = Thread.GetNamedDataSlot ("securityLevel");

  // Это свойство имеет отдельное значение в каждом потоке.
  int SecurityLevel
  {
    get
    {
      object data = Thread.GetData (_secSlot);
      return data == null ? 0 : (int) data;  // null == не инициализировано
    }
    set { Thread.SetData (_secSlot, value); }
  }
  ...
```

В показанном примере мы вызываем метод Thread.GetNamedDataSlot, который создает именованную ячейку – это позволяет разделять данную ячейку в рамках всего приложения. В качестве альтернативы можно самостоятельно управлять областью видимости ячейки посредством неименованной ячейки, получаемой с помощью вызова метода Thread.AllocateDataSlot:

```
class Test
{
  LocalDataStoreSlot _secSlot = Thread.AllocateDataSlot();
  ...
```

Метод Thread.FreeNamedDataSlot освободит именованную ячейку данных во всех потоках, но только если все ссылки на объект LocalDataStoreSlot покинули области видимости и были обработаны сборщиком мусора. Потоки не потеряют свои ячейки данных, если будут хранить ссылку на соответствующий объект LocalDataStoreSlot до тех пор, пока ячейка нужна.

# Interrupt и Abort

Методы Interrupt и Abort действуют вытесняющим образом на другой поток. Метод Interrupt не имеет допустимых сценариев использования, тогда как метод Abort изредка полезен.

Метод Interrupt принудительно освобождает заблокированный поток, генерируя в нем исключение ThreadInterruptedException. Если поток не блокирован, то выполнение продолжается до его следующего блокирования, после чего генерируется исключение ThreadInterruptedException. Метод Interrupt бесполезен, т.к. отсутствуют сценарии, для которых невозможно было бы построить лучшее решение с помощью сигнализирующих конструкций и признаков отмены (или метода Abort). Он также по своей сути опасен, поскольку никогда нельзя иметь уверенность, в каком месте кода поток окажется принудительно деблокированным (это может произойти внутри кода самой инфраструктуры .NET Framework, например).

Метод Abort пытается принудительно закончить другой поток, генерируя исключение ThreadAbortException в потоке прямо там, где он выполняется (кроме неуправляемого кода). Исключение ThreadAbortException необычно тем, что хотя его можно перехватить, оно генерируется повторно в конце блока catch (в попытке нормально завершить поток), если только не вызвать внутри блока catch метод Thread.ResetAbort. (В промежутке между этими моментами поток имеет состояние ThreadState, соответствующее AbortRequested.)

Необработанное исключение ThreadAbortException является одним из двух типов исключений, которые не приводят к завершению приложения (второй тип — AppDomainUnloadException).

Для предохранения целостности домена приложения учитываются любые блоки finally, а статические конструкторы никогда не прекращаются на середине своего выполнения. С учетом этого метод Abort не подходит для реализации универсальной отмены, т.к. существует возможность того, что прекращенный поток вызовет проблемы и нарушит работу домена приложения (или даже процесса).

Например, предположим, что конструктор экземпляров типа получает неуправляемый ресурс (скажем, файловый дескриптор), который освобождается в методе Dispose типа. Если поток прекращается до полного завершения конструктора, то частично сконструированный объект не сможет быть освобожден и произойдет утечка неуправляемого дескриптора. (Финализатор, если присутствует, по-прежнему запустится, но только чтобы сборщик мусора смог обработать объект.) Такая уязвимость характерна для базовых типов .NET Framework, включая FileStream, и делает метод Abort неподходящим в большинстве сценариев.

Расширенное обсуждение причин, по которым прекращение функционирования кода .NET Framework является небезопасным, можно найти в статье "Aborting Threads" ("Прекращение потоков") на нашем веб-сайте по адресу http://www.albahari.com/threading/.

Когда альтернатив применению метода Abort нет, смягчить бо́льшую часть потенциального разрушения можно, запустив поток в другом домене приложения и воссоздав текущий домен после прекращения работы потока (именно так поступает LINQPad в случае отмены запроса). Домены приложений обсуждаются в главе 24.

Вполне допустимо и безопасно вызывать метод Abort на собственном потоке, поскольку вы точно знаете, где находитесь. Иногда это удобно, когда нужно, чтобы исключение генерировалось повторно после каждого блока catch — в точности так поступает инфраструктура ASP.NET при вызове вами метода Redirect.

## Suspend и Resume

Методы Suspend и Resume замораживают и размораживают другой поток. Замороженный поток действует так, будто бы он заблокирован, хотя приостановка считается отличающейся от блокирования (как сообщает свойство ThreadState потока). Подобно Interrupt методы Suspend и Resume не имеют допустимых сценариев использования и потенциально опасны: если вы приостановите поток, когда он удерживает блокировку, то другие потоки не смогут получить данную блокировку (включая ваш поток), делая программу уязвимой к взаимоблокировкам. По указанной причине в .NET Framework 2.0 методы Suspend и Resume были объявлены не рекомендуемыми.

Тем не менее, приостановка потока обязательна, если необходимо получить трассировку стека в другом потоке. Временами это полезно для диагностических целей и может делаться следующим образом:

```
StackTrace stackTrace;  // в System.Diagnostics
targetThread.Suspend();
try { stackTrace = new StackTrace (targetThread, true); }
finally { targetThread.Resume(); }
```

К сожалению, такой код уязвим к взаимоблокировкам, потому что получение трассировки стека само связано с получением блокировок из-за применения рефлексии. Проблему можно обойти, обеспечив вызов Resume внутри другого потока, если спустя, скажем, 200 миллисекунд первый поток остается в приостановленном состоянии (после чего можно предположить, что произошла взаимоблокировка). Разумеется, трассировка стека станет недействительной, но такой подход намного лучше, чем взаимоблокировка приложения:

```
StackTrace stackTrace = null;
var ready = new ManualResetEventSlim();

new Thread (() =>
{
  // Ограничитель для освобождения потока в случае возникновения взаимоблокировки:
  ready.Set();
  Thread.Sleep (200);
  try { targetThread.Resume(); } catch { }
}).Start();

ready.Wait();
targetThread.Suspend();
try { stackTrace = new StackTrace (targetThread, true); }
catch { /* Взаимоблокировка */ }
finally
{
  try { targetThread.Resume(); }
  catch { stackTrace = null;  /* Взаимоблокировка */  }
}
```

# Таймеры

Если некоторый метод необходимо выполнять многократно через регулярные интервалы, то проще всего прибегнуть к помощи *таймера*. Таймеры удобны и эффективны в плане использования ими памяти и других ресурсов, если сравнивать их с такими приемами, как показанный ниже:

```
new Thread (delegate() {
                          while (enabled)
                          {
                            DoSomeAction();
                            Thread.Sleep (TimeSpan.FromHours (24));
                          }
                        }).Start();
```

Здесь не только надолго связывается ресурс потока, но без написания дополнительного кода метод `DoSomeAction` будет вызываться в более позднее время каждый день. Проблемы подобного рода решаются с помощью таймеров.

Инфраструктура .NET Framework предлагает четыре таймера. Два из них являются универсальными многопоточными таймерами:

- `System.Threading.Timer`
- `System.Timers.Timer`

Другие два представляют собой специализированные однопоточные таймеры:

- `System.Windows.Forms.Timer` (таймер Windows Forms)
- `System.Windows.Threading.DispatcherTimer` (таймер WPF)

Многопоточные таймеры характеризуются большей мощностью, точностью и гибкостью; однопоточные таймеры безопаснее и удобнее для запуска простых задач, которые обновляют элементы управления Windows Forms либо элементы WPF.

## Многопоточные таймеры

Класс `System.Threading.Timer` представляет простейший многопоточный таймер: он имеет только конструктор и два метода (предмет восхищения для минималистов, к которым себя относят и авторы книги). В следующем примере таймер вызывает метод `Tick`, который выводит строку `"tick..."` спустя пять секунд и затем ежесекундно, пока пользователь не нажмет клавишу <Enter>:

```
using System;
using System.Threading;
class Program
{
  static void Main()
  {
    // Первый интервал составляет 5000 мс; последующие интервалы - 1000 мс.
    Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
    Console.ReadLine();
    tmr.Dispose();  // Это останавливает таймер и производит очистку.
  }
  static void Tick (object data)
  {
    // Это запускается в потоке из пула.
    Console.WriteLine (data);  // Выводит "tick..."
  }
}
```

Обсуждение освобождения многопоточных таймеров можно найти в разделе "Таймеры" главы 12.

Позже интервал таймера можно изменить, вызвав его метод `Change`. Если нужно, чтобы таймер запустился только раз, тогда в последнем аргументе конструктора следует указать `Timeout.Infinite`.

Инфраструктура .NET Framework предоставляет еще один класс таймера с тем же именем, но в пространстве имен `System.Timers`. Это просто оболочка для `System.Threading.Timer`, которая предлагает дополнительные удобства, но имеет идентичный внутренний механизм. Ниже приведена сводка по добавленным возможностям:

- реализация интерфейса `IComponent`, которая позволяет классу находиться в панели компонентов визуального редактора Visual Studio;
- свойство `Interval` вместо метода `Change`;
- событие `Elapsed` вместо делегата обратного вызова;
- свойство `Enabled` для запуска и останова таймера (стандартным значением является `false`);
- методы `Start` и `Stop` на тот случай, если вам не нравится работать со свойством `Enabled`;
- флаг `AutoReset` для указания повторяющегося события (стандартным значением является `true`);
- свойство `SynchronizingObject` с методами `Invoke` и `BeginInvoke` для безопасного вызова методов на элементах WPF и элементах управления Windows Forms.

Рассмотрим пример:

```
using System;
using System.Timers;  // Пространство имен Timers, а не Threading

class SystemTimer
{
  static void Main()
  {
    Timer tmr = new Timer();     // Не требует никаких аргументов
    tmr.Interval = 500;
    tmr.Elapsed += tmr_Elapsed;  // Использует событие вместо делегата
    tmr.Start();               // Запуск таймера
    Console.ReadLine();
    tmr.Stop();                // Останов таймера
    Console.ReadLine();
    tmr.Start();               // Повторный запуск таймера
    Console.ReadLine();
    tmr.Dispose();             // Останов таймера навсегда
  }
  static void tmr_Elapsed (object sender, EventArgs e)
  {
    Console.WriteLine ("Tick");
  }
}
```

Многопоточные таймеры применяют пул потоков, чтобы позволить нескольким потокам обслуживать множество таймеров. Это означает, что метод обратного вызова или событие `Elapsed` может инициироваться каждый раз в новом потоке, когда к нему производится обращение. Кроме того, событие `Elapsed` всегда инициируется (приблизительно) вовремя – независимо от того, завершило ли выполнение предыдущее событие `Elapsed`. Следовательно, обратные вызовы или обработчики событий должны быть безопасными в отношении потоков.

Точность многопоточных таймеров зависит от операционной системы и обычно находится в диапазоне 10–20 миллисекунд. Если нужна более высокая точность, можете прибегнуть к собственному взаимодействию и обратиться к мультимедиа-таймеру Windows. Его точность достигает 1 миллисекунды, а сам он определен в сборке `winmm.dll`. Сначала вызовите функцию `timeBeginPeriod`, чтобы проинформировать операционную систему о том, что необходима высокая точность измерения времени, а затем обратитесь к функции `timeSetEvent` для запуска мультимедиа-таймера. По завершении работы вызовите функцию `timeKillEvent`, чтобы остановить таймер, и функцию `timeEndPeriod` для сообщения операционной системе о том, что высокая точность измерения времени больше не нужна. Вызов внешних методов с помощью P/Invoke демонстрируется в главе 25. Полноценные примеры работы с мультимедиа-таймером можно найти в Интернете, выполнив поиск по ключевым словам *dllimport winmm.dll timesetevent*.

## Однопоточные таймеры

Инфраструктура .NET Framework предоставляет таймеры, которые предназначены для устранения проблем с безопасностью к потокам в приложениях WPF и Windows Forms:

- `System.Windows.Threading.DispatcherTimer` (WPF)
- `System.Windows.Forms.Timer` (Windows Forms)

Однопоточные таймеры не проектировались для работы за пределами соответствующих сред. Например, если попытаться использовать таймер Windows Forms в приложении Windows Service, то даже не будет инициировано событие таймера!

Оба однопоточных таймера похожи на `System.Timers.Timer` в плане открытых членов – `Interval`, `Start` и `Stop` (а также `Tick`, который эквивалентен `Elapsed`) – и применяются в аналогичной манере. Однако они отличаются своей внутренней работой. Вместо запуска событий таймера в потоках из пула они отправляют события циклу обработки сообщений WPF или Windows Forms. В результате событие `Tick` всегда инициируется в том же самом потоке, который первоначально создал таймер – в нормальном приложении это поток, используемый для управления всеми элементами пользовательского интерфейса. Такой подход обеспечивает несколько преимуществ:

- вы можете вообще забыть о безопасности к потокам;
- новый вызов `Tick` никогда не будет инициирован до тех пор, пока предыдущий вызов `Tick` не завершит обработку;
- обновлять элементы управления пользовательского интерфейса можно напрямую из кода обработки события `Tick`, не вызывая `Control.BeginInvoke` или `Dispatcher.BeginInvoke`.

Таким образом, программа, эксплуатирующая такие таймеры, в действительности не является многопоточной: в итоге получается та же разновидность псевдопараллелизма, которая была описана в главе 14 при рассмотрении асинхронных функций, выполняющихся в потоке пользовательского интерфейса. Один поток обслуживает все таймеры – равно как и обрабатывает события пользовательского интерфейса. Это значит, что обработчик события `Tick` должен выполняться быстро, иначе пользовательский интерфейс станет неотзывчивым.

Следовательно, таймеры WPF и Windows Forms подходят для выполнения небольших работ, обычно связанных с обновлением какого-то аспекта пользовательского интерфейса (например, часов или счетчика с обратным отсчетом).

В терминах точности однопоточные таймеры похожи на многопоточные таймеры (десятки миллисекунд), хотя они обычно менее *точны*, поскольку могут задерживаться на время, пока обрабатываются другие запросы пользовательского интерфейса (или другие события таймеров).

# 23

# Параллельное программирование

В настоящей главе мы раскроем многопоточные API-интерфейсы и конструкции, направленные на использование преимуществ многоядерных процессоров:

- параллельный LINQ (Parallel LINQ), или PLINQ;
- класс `Parallel`;
- конструкции параллелизма задач;
- параллельные коллекции.

Все перечисленное было добавлено в версии .NET Framework 4.0 и вместе известно под (свободным) названием PFX (Parallel Framework). Класс `Parallel` и конструкции параллелизма задач называют *библиотекой параллельных задач* (Task Parallel Library – TPL).

Чтение главы требует знания основ, изложенных в главе 14, в частности блокирования, безопасности к потокам и класса `Task`.

## Для чего нужна инфраструктура PFX?

На протяжении последнего десятилетия производители центральных процессоров (ЦП) перешли с одноядерной на многоядерную архитектуру. В результате создается дополнительная проблема для нас как программистов, поскольку однопоточный код не будет автоматически выполняться быстрее только по причине наличия дополнительных ядер.

Использовать в своих интересах множество ядер довольно легко в большинстве серверных приложений, где каждый поток может независимо обрабатывать отдельный клиентский запрос, но труднее в настольных приложениях, т.к. обычно это требует применения к коду с интенсивными вычислениями следующих действий.

1. *Разбиение* кода на небольшие части.
2. Выполнение частей кода параллельно через многопоточность.
3. *Объединение* результатов по мере их получения в безопасной к потокам и высокопроизводительной манере.

Хотя все указанные действия можно реализовать с помощью классических многопоточных конструкций, выполнять их довольно утомительно – особенно шаги разбиения и объединения. Еще одна проблема связана с тем, что обычная стратегия блокирования для обеспечения безопасности в отношении потоков приводит к большому числу состязаний, когда множество потоков одновременно работают с одними и теми же данными.

Библиотеки PFX были спроектированы специально для оказания помощи в таких сценариях.

Программирование с целью получения выгоды от множества ядер или процессоров называют *параллельным программированием*. Оно представляет собой подмножество более широкой концепции многопоточности.

## Концепции PFX

Существуют две стратегии разбиения работы между потоками: *параллелизм данных* и *параллелизм задач*.

Когда набор задач должен быть выполнен над множеством значений данных, мы можем распараллелить работу, заставив каждый поток выполнять (тот же самый) набор задач на подмножестве значений. Это называется *параллелизмом данных*, потому что мы распределяем *данные* между потоками. Напротив, при *параллелизме задач* мы распределяем *задачи*; другими словами, заставляем каждый поток выполнять свою задачу.

В общем случае параллелизм данных реализуется легче и масштабируется лучше для оборудования с высокой степенью параллелизма, т.к. он сокращает или устраняет разделяемые данные (и тем самым сводит к минимуму проблемы, связанные с состязаниями и безопасностью к потокам). Кроме того, параллелизм данных задействует тот факт, что значений данных часто имеется больше, чем дискретных задач, увеличивая в итоге потенциал параллелизма.

Параллелизм данных также способствует *структурированному параллелизму*, который означает, что параллельные единицы работы начинаются и завершаются в одном и том же месте внутри программы. В отличие от него параллелизм задач имеет тенденцию быть неструктурированным, т.е. параллельные единицы работы могут начинаться и завершаться в разных местах, разбросанных по программе. Структурированный параллелизм проще, менее подвержен ошибкам и позволяет поручить выполнение сложной работы по разбиению и координации потоков (и даже объединение результатов) библиотекам.

## Компоненты PFX

Как показано на рис. 23.1, инфраструктура PFX содержит два уровня функциональности. Верхний уровень состоит из двух API-интерфейсов *структурированного параллелизма данных*: PLINQ и класс `Parallel`. Нижний уровень включает классы параллелизма задач, а также набор дополнительных конструкций, помогающих выполнять действия параллельного программирования.

Язык PLINQ предлагает самую развитую функциональность: он автоматизирует все шаги по распараллеливанию – включая разбиение работы на задачи, выполнение этих задач в потоках и объединение результатов в единственную выходную последовательность. Он называется *декларативным*, поскольку вы просто декларируете, что хотите распараллелить свою работу (структурируя ее как запрос LINQ), и позволяете инфраструктуре .NET Framework позаботиться о деталях реализации.

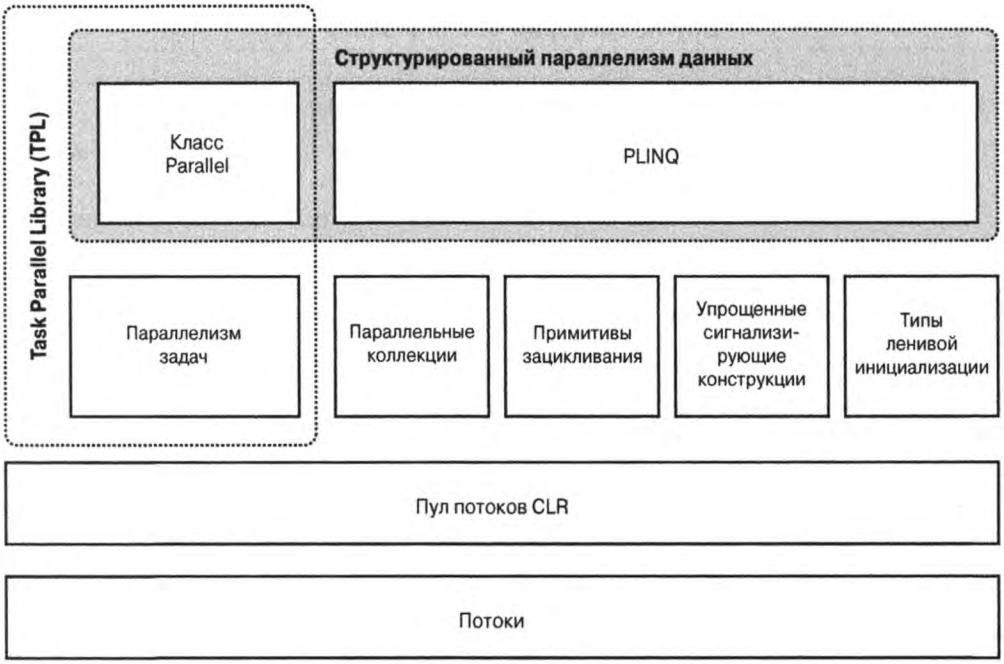



***Рис. 23.1.** Компоненты PFX*

Напротив, другие подходы являются *императивными* в том, что вы должны явно писать код для разбиения и объединения. В случае класса `Parallel` вам придется объединять результаты самостоятельно; имея дело с конструкциями параллелизма задач, самостоятельно реализовывать придется также и разбиение работы:

| | Разбивает работу | Объединяет результаты |
|---|---|---|
| PLINQ | Да | Да |
| Класс `Parallel` | Да | Нет |
| Параллелизм задач PFX | Нет | Нет |

Параллельные коллекции и примитивы зацикливания помогают справиться с действиями параллельного программирования нижнего уровня. Они важны из-за того, что инфраструктура PFX спроектирована для работы не только с современным оборудованием, но и с будущими поколениями процессоров с намного бо́льшим числом ядер. Если вы хотите перенести штабель бревен и для этого у вас есть 32 рабочих, то самой сложной проблемой будет обеспечение таких условий, при которых рабочие не мешали бы друг другу. То же самое касается разбиения алгоритма по 32 ядрам: если для защиты общих ресурсов применяются обычные блокировки, то результирующая блокировка может означать, что на самом деле одновременно занятыми является только некоторая доля ядер. Параллельные коллекции настраиваются специально для доступа с высокой степенью параллелизма, преследуя цель свести к минимуму или вообще устранить блокирование. Язык PLINQ и класс `Parallel` сами полагаются на параллельные коллекции и на примитивы зацикливания для эффективного управления работой.

### Другие сценарии использования PFX

Конструкции параллельного программирования полезны не только для работы с многоядерными процессорами, но также и в других сценариях.

- Параллельные коллекции иногда подходят, когда нужна безопасная к потокам очередь, стек или словарь.
- Класс `BlockingCollection` предоставляет простые средства для реализации структур производителей/потребителей и является хорошим способом ограничения параллелизма.
- Задачи являются основой асинхронного программирования, как было показано в главе 14.

## Когда необходимо использовать инфраструктуру PFX?

Основным сценарием использования PFX является *параллельное программирование*: привлечение множества процессорных ядер, чтобы ускорить выполнение интенсивного в плане вычислений кода.

Сложность с применением многоядерных процессоров обусловлена законом Амдала, который утверждает, что максимальное улучшение производительности от распараллеливания определяется той частью кода, которая должна выполняться последовательно. Например, если хотя бы две трети времени выполнения алгоритма поддаются распараллеливанию, то никогда не удастся превысить трехкратный выигрыш в производительности – даже при неограниченном числе ядер.

Таким образом, прежде чем продолжать, полезно проверить, что узкое место находится в распараллеливаемом коде. Также имеет смысл обдумать, *должен* ли ваш код действительно быть интенсивным в плане вычислений – часто простейшим и наиболее эффективным подходом будет оптимизация. Тем не менее, следует соблюдать компромисс, потому что некоторые технологии оптимизации могут затруднить распараллеливание кода.

Самый простой выигрыш получается с так называемыми *естественно параллельными* случаями – когда работа может быть легко разбита на задачи, которые сами по себе выполняются эффективно (здесь очень хорошо подходит структурированный параллелизм). Примеры включают многие задачи обработки изображений, метод трассировки лучей и прямолинейные подходы в математике или криптографии. Примером неестественной параллельной задачи может считаться реализация оптимизированной версии алгоритма быстрой сортировки – хороший результат требует некоторых размышлений и возможно неструктурированного параллелизма.

# PLINQ

Инфраструктура PLINQ автоматически распараллеливает локальные запросы LINQ. Преимущество PLINQ заключается в простоте использования, т.к. ответственность за выполнение работ по разбиению и объединению результатов перекладывается на .NET Framework.

Для применения PLINQ просто вызовите метод `AsParallel` на входной последовательности и затем продолжайте запрос LINQ обычным образом. Приведенный ниже запрос вычисляет простые числа между 3 и 100 000, обеспечивая полную загрузку всех ядер процессора на целевой машине:

```
// Вычислить простые числа с использованием простого (неоптимизированного) алгоритма.
IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);
var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;
int[] primes = parallelQuery.ToArray();
```

AsParallel представляет собой расширяющий метод в классе System.Linq.ParallelEnumerable. Он помещает входные данные в оболочку последовательности, основанной на ParallelQuery<TSource>, что вызывает привязку последующих операций запросов LINQ к альтернативному набору расширяющих методов, которые определены в классе ParallelEnumerable. Они предоставляют параллельные реализации для всех стандартных операций запросов. По существу они разбивают входную последовательность на порции, которые выполняются в разных потоках, и объединяют результаты снова в единственную выходную последовательность для дальнейшего потребления (рис. 23.2).

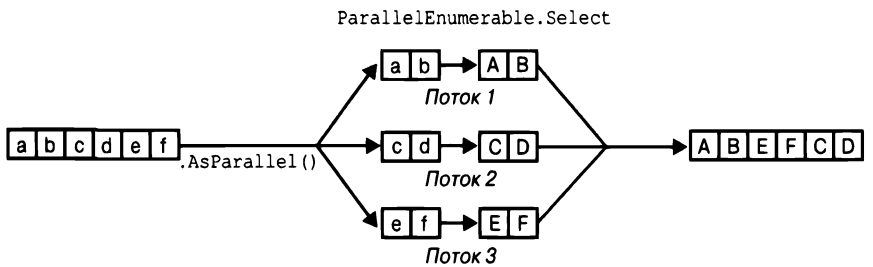



***Рис. 23.2.** Модель выполнения PLINQ*

Вызов метода AsSequential извлекает последовательность из оболочки ParallelQuery, так что дальнейшие операции запросов привязываются к стандартному набору операций и выполняются последовательно. Такое действие нужно предпринимать перед вызовом методов, которые имеют побочные эффекты или не являются безопасными в отношении потоков.

Для операций запросов, принимающих две входные последовательности (Join, GroupJoin, Concat, Union, Intersect, Except и Zip), метод AsParallel должен быть применен к обеим входным последовательностям (иначе сгенерируется исключение). Однако по мере продвижения запроса применять к нему AsParallel нет необходимости, т.к. операции запросов PLINQ выводят еще одну последовательность ParallelQuery. На самом деле дополнительный вызов AsParallel привносит неэффективность, связанную с тем, что он инициирует слияние и повторное разбиение запроса:

```
mySequence.AsParallel()        // Помещает последовательность
                               // в оболочку ParallelQuery<int>
          .Where (n => n > 100) // Выводит другую последовательность
                               // ParallelQuery<int>
          .AsParallel()        // Необязательно — и неэффективно!
          .Select (n => n * n)
```

Не все операции запросов можно эффективно распараллеливать. Для операций, не поддающихся распараллеливанию (см. раздел "Ограничения PLINQ" далее в главе), PLINQ взамен реализует последовательное выполнение. Инфраструктура PLINQ может также оперировать последовательно, если ожидает, что накладные расходы от распараллеливания в действительности замедлят определенный запрос.

Инфраструктура PLINQ предназначена только для локальных коллекций: она не работает с LINQ to SQL или Entity Framework, потому что в таких ситуациях LINQ транслируется в код SQL, который затем выполняется на сервере баз данных. Тем не менее, PLINQ *можно* использовать для выполнения дополнительных локальных запросов в результирующих наборах, полученных из запросов к базам данных.

Если запрос PLINQ генерирует исключение, то оно повторно генерируется как объект `AggregateException`, свойство `InnerExceptions` которого содержит реальное исключение (или исключения). Дополнительные сведения можно найти в разделе "Работа с `AggregateException`" далее в главе.

### Почему метод `AsParallel` не выбран в качестве варианта по умолчанию?

Учитывая, что метод `AsParallel` прозрачно распараллеливает запросы LINQ, возникает вопрос: почему в Microsoft решили не распараллеливать стандартные операции запросов, сделав PLINQ вариантом по умолчанию?

Есть несколько причин выбора подхода с *включением*. Первая причина связана с тем, что для получения пользы от PLINQ в наличии должен быть обоснованный объем работы с интенсивными вычислениями, которую можно было бы поручить рабочим потокам. Большинство потоков LINQ to Objects выполняются очень быстро, и распараллеливание для них не только окажется излишним, но накладные расходы на разбиение, объединение и координацию дополнительных потоков фактически могут даже замедлить их выполнение.

Ниже перечислены другие причины.

- Вывод запроса PLINQ (по умолчанию) может отличаться от вывода запроса LINQ в том, что касается порядка следования элементов (как объясняется в разделе "PLINQ и упорядочивание" далее в главе).
- PLINQ помещает исключения в оболочку `AggregateException` (чтобы учесть возможность генерации множества исключений).
- PLINQ будет давать ненадежные результаты, если запрос вызывает небезопасные к потокам методы.

Наконец, PLINQ предлагает немало способов настройки. Обременение стандартного API-интерфейса LINQ to Objects нюансами такого рода добавило бы путаницы.

## Продвижение параллельного выполнения

Подобно обычным запросам LINQ запросы PLINQ оцениваются ленивым образом. Другими словами, выполнение будет инициировано, только когда начнется потребление результатов – как правило, посредством цикла `foreach` (хотя оно также может происходить через операцию преобразования, такую как `ToArray`, или операцию, которая возвращает одиночный элемент либо значение).

Тем не менее, при перечислении результатов выполнение продолжается несколько иначе, чем в случае обычного последовательного запроса. Последовательный запрос поддерживается полностью потребителем с применением модели с пассивным источником: каждый элемент из входной последовательности извлекается только тогда, когда он затребован потребителем. Параллельный запрос обычно использует независимые потоки для извлечения элементов из входной последовательности, причем с небольшим *упреждением*, до того момента, когда они понадобятся потребителю (почти как телесуфлер у дикторов новостей или буфер в проигрывателях компакт-дисков). Затем он обрабатывает элементы параллельно через цепочку запросов, удерживая результаты в небольшом буфере, чтобы они были готовы при затребовании потребителем. Если потребитель приостанавливает или прекращает перечисление до его завершения, обработчик запроса также приостанавливается или прекращает работу, чтобы не тратить впустую время ЦП или память.

Поведение буферизации PLINQ можно настраивать, вызывая метод `WithMergeOptions` после `AsParallel`. Стандартное значение `AutoBuffered` перечисления `ParallelMergeOptions` обычно дает наилучшие окончательные результаты. Значение `NotBuffered` отключает буфер и полезно в ситуации, когда результаты необходимо увидеть как можно скорее; значение `FullyBuffered` кеширует целый результирующий набор перед представлением его потребителю (подобным образом изначально работают операции `OrderBy` и `Reverse`, а также операции над элементами, операции агрегирования и операции преобразования).

## PLINQ и упорядочивание

Побочный эффект от распараллеливания операций запросов заключается в том, что когда результаты объединены, они не обязательно находятся в том же самом порядке, в котором они были получены (см. рис. 23.2). Другими словами, обычная гарантия предохранения порядка LINQ для последовательностей больше не поддерживается.

Если нужно предохранение порядка, тогда после вызова `AsParallel` понадобится вызвать метод `AsOrdered`:

```
myCollection.AsParallel().AsOrdered()...
```

Вызов метода `AsOrdered` оказывает влияние на производительность, поскольку инфраструктура PLINQ должна отслеживать исходные позиции всех элементов.

Позже последствия от вызова `AsOrdered` в запросе можно отменить, вызвав метод `AsUnordered`: это вводит "точку случайного тасования", которая позволяет запросу выполняться более эффективно после ее прохождения. Таким образом, если необходимо предохранить упорядочение входной последовательности только для первых двух операций запросов, то можно поступить так:

```
inputSequence.AsParallel().AsOrdered()
  .QueryOperator1()
  .QueryOperator2()
  .AsUnordered()  // Начиная с этой точки, упорядочивание роли не играет
  .QueryOperator3()
  ...
```

Метод `AsOrdered` не является стандартным вариантом, потому что для большинства запросов первоначальное упорядочивание во входной последовательности

не имеет значения. Другими словами, если бы метод AsOrdered использовался по умолчанию, то к большинству параллельных запросов пришлось бы применять метод AsUnordered, чтобы добиться лучших показателей производительности, и поступать так было бы обременительно.

## Ограничения PLINQ

Существует несколько практических ограничений относительно того, что инфраструктура PLINQ способна распараллеливать. Ограничения могут быть ослаблены в последующих пакетах обновлений и версиях инфраструктуры .NET Framework.

Следующие операции запросов предотвращают распараллеливание запроса, если только исходные элементы не находятся в своих первоначальных индексных позициях:

- индексированные версии Select, SelectMany и ElementAt.

Большинство операций запросов изменяют индексные позиции элементов (включая операции, удаляющие элементы, такие как Where). Это означает, что если нужно использовать предшествующие операции, то они обычно должны располагаться в начале запроса.

Следующие операции запросов допускают распараллеливание, но применяют затратную стратегию разбиения, которая иногда может оказываться медленнее последовательной обработки:

- Join, GroupBy, GroupJoin, Distinct, Union, Intersect и Except.

Перегруженные версии операции Aggregate, принимающие начальное значение (в аргументе seed), в их стандартном виде не поддерживают возможность распараллеливания – в PLINQ для такой цели предлагаются специальные перегруженные версии (см. раздел "Оптимизация PLINQ" далее в главе).

Все остальные операции поддаются распараллеливанию, хотя их использование не гарантирует, что запрос будет распараллелен. Инфраструктура PLINQ может выполнять запрос последовательно, если ожидает, что накладные расходы от распараллеливания приведут к замедлению имеющегося конкретного запроса. Такое поведение можно переопределить и принудительно применять параллелизм, вызвав показанный ниже метод после AsParallel:

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

## Пример: параллельная программа проверки орфографии

Предположим, что требуется написать программу проверки орфографии, которая выполняется быстро для очень больших документов за счет использования всех свободных процессорных ядер. Выразив алгоритм в виде запроса LINQ, мы легко можем его распараллелить.

Первый шаг предусматривает загрузку словаря английских слов в объект HashSet, чтобы обеспечить эффективный поиск:

```
if (!File.Exists ("WordLookup.txt"))  // Содержит около 150 000 слов
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);
```

Затем мы будем применять полученное средство поиска слов для создания тестового "документа", содержащего массив из миллиона случайных слов. После построения массива мы внесем пару орфографических ошибок:

```
var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";  // Внесение пары
wordsToTest [23456] = "wubsie";  // орфографических ошибок.
```

Теперь мы можем выполнить параллельную проверку орфографии, сверяя wordsToTest с wordLookup. PLINQ позволяет делать это очень просто:

```
var query = wordsToTest
  .AsParallel()
  .Select  ((word, index) => new IndexedWord { Word=word, Index=index })
  .Where   (iword => !wordLookup.Contains (iword.Word))
  .OrderBy (iword => iword.Index);

foreach (var mistake in query)
  Console.WriteLine (mistake.Word + " - index = " + mistake.Index);

// ВЫВОД:
// woozsh - index = 12345
// wubsie - index = 23456
```

IndexedWord — это специальная структура, которая определена следующим образом:

```
struct IndexedWord { public string Word; public int Index; }
```

Метод wordLookup.Contains в предикате придает запросу определенный "вес" и делает уместным его распараллеливание.

Мы могли бы слегка упростить запрос за счет использования анонимного типа вместо структуры IndexedWord. Однако это привело бы к снижению производительности, т.к. анонимные типы (будучи классами, а потому ссылочными типами) привносят накладные расходы на выделение памяти в куче и последующую сборку мусора.

Разница может оказаться недостаточной, чтобы иметь значение в последовательных запросах, но в случае параллельных запросов весьма выгодно отдавать предпочтение выделению памяти в стеке. Причина в том, что выделение памяти в стеке хорошо поддается распараллеливанию (поскольку каждый поток имеет собственный стек), в то время как в противном случае все потоки должны состязаться за одну и ту же кучу, управляемую единственным диспетчером памяти и сборщиком мусора.

### Использование ThreadLocal<T>

Давайте расширим наш пример, распараллелив само создание случайного тестового списка слов. Мы структурировали его как запрос LINQ, так что все должно быть легко. Вот последовательная версия:

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();
```

К сожалению, вызов метода random.Next небезопасен в отношении потоков, поэтому работа не сводится к простому добавлению в запрос вызова AsParallel. Потенциальным решением может быть написание функции, помещающей вызов random.Next внутрь блокировки, но это ограничило бы параллелизм. Более удачный вариант предусматривает применение класса ThreadLocal<Random> (см. раздел "Локальное хранилище потока" в главе 22) с целью создания отдельного объекта Random для каждого потока. Тогда распараллелить запрос можно следующим образом:

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );
string[] wordsToTest = Enumerable.Range (0, 1000000).AsParallel()
  .Select (i => wordList [localRandom.Value.Next (0, wordList.Length)])
  .ToArray();
```

В нашей фабричной функции для создания объекта Random мы передаем хеш-код Guid, гарантируя тем самым, что даже если два объекта Random создаются в рамках короткого промежутка времени, то они все равно будут выдавать отличающиеся последовательности случайных чисел.

### Когда необходимо использовать PLINQ?

Довольно заманчиво поискать в существующих приложениях запросы LINQ и поэкспериментировать с их распараллеливанием. Однако обычно это непродуктивно, т.к. большинство задач, для которых LINQ является очевидным лучшим решением, выполняются очень быстро, а потому не выигрывают от распараллеливания. Более удачный подход предполагает поиск узких мест, интенсивно использующих ЦП, и выяснение, могут ли они быть выражены в виде запроса LINQ. (Приятный побочный эффект от такой реструктуризации состоит в том, что LINQ обычно делает код более кратким и читабельным.)

Инфраструктура PLINQ хорошо подходит для естественно параллельных задач. Однако она может быть плохим выбором для обработки изображений, потому что объединение миллионов пикселей в выходную последовательность создаст узкое место. Взамен пиксели лучше записывать прямо в массив или блок неуправляемой памяти и применять класс Parallel либо параллелизм задач для управления многопоточностью. (Тем не менее, объединение результатов можно аннулировать с использованием ForAll — мы обсудим данную тему в разделе "Оптимизация PLINQ" далее в главе. Поступать так имеет смысл, если алгоритм обработки изображений естественным образом приспосабливается к LINQ.)

## Функциональная чистота

Поскольку PLINQ запускает ваш запрос в параллельных потоках, вы должны избегать выполнения небезопасных к потокам операций. В частности, запись в переменные порождает *побочные эффекты*, следовательно, она не является безопасной в отношении потоков:

```
// Следующий запрос умножает каждый элемент на его позицию.
// Получив на входе Enumerable.Range(0,999), он должен
// вывести последовательность квадратов.
int i = 0;
var query = from n in Enumerable.Range(0,999).AsParallel() select n * i++;
```

Мы могли бы сделать инкрементирование переменной i безопасным к потокам за счет применения блокировок, но все еще останется проблема того, что i не обязательно будет соответствовать позиции входного элемента. И добавление AsOrdered в запрос не решит последнюю проблему, т.к. метод AsOrdered гарантирует лишь то, что элементы выводятся в порядке, согласованном с порядком, который бы они имели при последовательной обработке — он не осуществляет действительную их *обработку* последовательным образом.

Взамен данный запрос должен быть переписан с использованием индексированной версии Select:

```
var query = Enumerable.Range(0,999).AsParallel().Select ((n, i) => n * i);
```

Для достижения лучшей производительности любые методы, вызываемые из операций запросов, должны быть безопасными к потокам, не производя запись в поля или свойства (не давать побочные эффекты, т.е. быть *функционально чистыми*). Если они являются безопасными в отношении потоков благодаря блокированию, тогда потенциал параллелизма запроса будет ограничен продолжительностью действия блокировки, деленной на общее время, которое занимает выполнение данной функции.

## Установка степени параллелизма

По умолчанию PLINQ выбирает оптимальную степень параллелизма для задействованного процессора. Ее можно переопределить, вызвав метод WithDegreeOfParallelism после AsParallel:

```
...AsParallel().WithDegreeOfPallelism(4)...
```

Примером, когда степень параллелизма может быть увеличена до значения, превышающего количество ядер, является работа с интенсивным вводом-выводом (скажем, загрузка множества веб-страниц за раз). Тем не менее, начиная с версии .NET Framework 4.5, комбинаторы задач и асинхронные функции предлагают аналогично несложное, но более *эффективное* решение (см. раздел "Комбинаторы задач" в главе 14). В отличие от объектов Task, инфраструктура PLINQ не способна выполнять работу с интенсивным вводом-выводом без блокирования потоков (и что еще хуже — потоков *из пула*).

### Изменение степени параллелизма

Метод WithDegreeOfParallelism можно вызывать только один раз внутри запроса PLINQ. Если его необходимо вызвать снова, то потребуется принудительно инициировать слияние и повторное разбиение запроса, еще раз вызвав метод AsParallel внутри запроса:

```
"The Quick Brown Fox"
  .AsParallel().WithDegreeOfParallelism (2)
  .Where (c => !char.IsWhiteSpace (c))
  .AsParallel().WithDegreeOfParallelism (3) //Инициировать слияние и разбиение
  .Select (c => char.ToUpper (c))
```

## Отмена

Отменить запрос PLINQ, результаты которого потребляются в цикле foreach, легко: нужно просто прекратить цикл foreach и запрос будет автоматически отменен по причине неявного освобождения перечислителя.

Отменить запрос, который заканчивается операцией преобразования, операцией над элементами или операцией агрегирования, можно из другого потока через признак отмены (см. раздел "Отмена" в главе 14). Чтобы вставить такой признак, необходимо после вызова AsParallel вызвать метод WithCancellation, передав ему свойство Token объекта CancellationTokenSource. Затем другой поток может вызвать метод Cancel на источнике признака, что приведет к генерации исключения OperationCanceledException в потребителе запроса:

```
IEnumerable<int> million = Enumerable.Range (3, 1000000);
var cancelSource = new CancellationTokenSource();
var primeNumberQuery =
  from n in million.AsParallel().WithCancellation (cancelSource.Token)
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;
new Thread (() => {
                    Thread.Sleep (100);       // Отменить запрос по
                    cancelSource.Cancel();    // прошествии 100 мс.
                  }
           ).Start();
try
{
  // Начать выполнение запроса:
  int[] primes = primeNumberQuery.ToArray();
  // Мы никогда не попадем сюда, потому что другой поток инициирует отмену.
}
catch (OperationCanceledException)
{
  Console.WriteLine ("Query canceled");     // Запрос отменен
}
```

Инфраструктура PLINQ не прекращает потоки вытесняющим образом из-за связанной с этим опасности (см. раздел "Interrupt и Abort" в главе 22). Взамен при инициировании отмены она ожидает завершения каждого рабочего потока со своим текущим элементом перед тем, как закончить запрос. Это означает, что любые внешние методы, которые вызывает запрос, будут выполняться до полного завершения.

# Оптимизация PLINQ

## Оптимизация на выходной стороне

Одно из преимуществ инфраструктуры PLINQ связано с тем, что она удобно объединяет результаты распараллеленной работы в единую выходную последовательность. Однако иногда все, что в итоге делается с такой последовательностью – выполнение некоторой функции над каждым элементом:

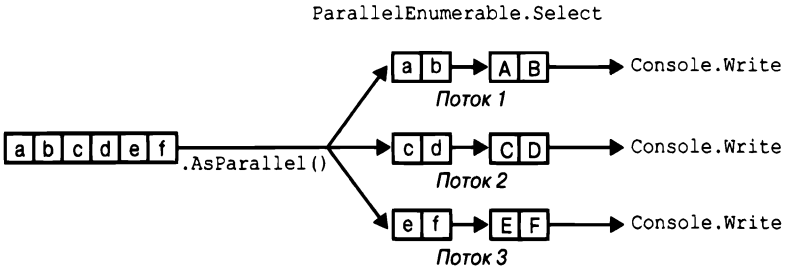
```
foreach (int n in parallelQuery)
  DoSomething (n);
```

В таком случае, если порядок обработки элементов не волнует, тогда эффективность можно улучшить с помощью метода ForAll из PLINQ.

Метод ForAll запускает делегат для каждого выходного элемента ParallelQuery. Он проникает прямо внутрь PLINQ, обходя шаги объединения и перечисления результатов. Ниже приведен простейший пример:

```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write);
```

Процесс продемонстрирован на рис. 23.3.




```
"abcdef" .AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write)
```

***Рис. 23.3.*** *Метод* `ForAll` *из PLINQ*

Объединение и перечисление результатов — не массовая затратная операция, поэтому оптимизация с помощью `ForAll` дает наибольшую выгоду при наличии большого количества быстро обрабатываемых входных элементов.

## Оптимизация на входной стороне

Для назначения входных элементов потокам в PLINQ поддерживаются три стратегии разбиения:

| Стратегия | Распределение элементов | Относительная производительность |
|---|---|---|
| Разбиение на основе порций | Динамическое | Средняя |
| Разбиение на основе диапазонов | Статическое | От низкой до очень высокой |
| Разбиение на основе хеш-кодов | Статическое | Низкая |

Для операций запросов, которые требуют сравнения элементов (`GroupBy`, `Join`, `GroupJoin`, `Intersect`, `Except`, `Union` и `Distinct`), выбор отсутствует: PLINQ всегда использует *разбиение на основе хеш-кодов*. Разбиение на основе хеш-кодов относительно неэффективно в том, что оно требует предварительного вычисления хеш-кода каждого элемента (а потому элементы с одинаковыми хеш-кодами могут обрабатываться в одном и том же потоке). Если вы сочтете это слишком медленным, то единственно доступным вариантом будет вызов метода `AsSequential` с целью отключения распараллеливания.

Для всех остальных операций запросов имеется выбор между разбиением на основе диапазонов и разбиением на основе порций. По умолчанию:

- если входная последовательность индексируема (т.е. является массивом или реализует интерфейс `IList<T>`), тогда PLINQ выбирает разбиение на основе диапазонов;
- иначе PLINQ выбирает разбиение на основе порций.

По своей сути разбиение на основе диапазонов выполняется быстрее с длинными последовательностями, для которых каждый элемент требует сходного объема времени ЦП на обработку. В противном случае разбиение на основе порций обычно быстрее.

Чтобы принудительно применить *разбиение на основе диапазонов*, выполните такие действия:

- если запрос начинается с вызова метода `Enumerable.Range`, то замените его вызовом `ParallelEnumerable.Range`;
- иначе просто вызовите метод `ToList` или `ToArray` на входной последовательности (очевидно, это повлияет на производительность, что также должно приниматься во внимание).

Метод `ParallelEnumerable.Range` — не просто сокращение для вызова `Enumerable.Range(...).AsParallel()`. Он изменяет производительность запроса, активизируя разбиение на основе диапазонов.

Чтобы принудительно применить *разбиение на основе порций*, необходимо поместить входную последовательность в вызов `Partitioner.Create` (из пространства имен `System.Collection.Concurrent`) следующим образом:

```
int[] numbers = { 3, 4, 5, 6, 7, 8, 9 };
var parallelQuery =
  Partitioner.Create (numbers, true).AsParallel()
  .Where (...)
```

Второй аргумент `Partitioner.Create` указывает на то, что для запроса требуется *балансировка загрузки*, которая представляет собой еще один способ сообщения о выборе разбиения на основе порций.

Разбиение на основе порций работает путем предоставления каждому рабочему потоку возможности периодически захватывать из входной последовательности небольшие "порции" элементов с целью их обработки (рис. 23.4). Инфраструктура PLINQ начинает с выделения очень маленьких порций (один или два элемента за раз) и затем по мере продвижения запроса увеличивает размер порции: это гарантирует, что небольшие последовательности будут эффективно распараллеливаться, а крупные последовательности не приведут к чрезмерным циклам полного обмена. Если рабочий поток получает "простые" элементы (которые обрабатываются быстро), то в конечном итоге он сможет получить больше порций. Такая система сохраняет каждый поток одинаково занятым (а процессорные ядра "сбалансированными"); единственный недостаток состоит в том, что извлечение элементов из разделяемой входной последовательности требует синхронизации (обычно монопольной блокировки) — и в результате могут появиться некоторые накладные расходы и состязания.

Разбиение на основе диапазонов пропускает обычное перечисление на входной стороне и предварительно распределяет одинаковое количество элементов для каждого рабочего потока, избегая состязаний на входной последовательности. Но если случится так, что некоторые потоки получат простые элементы и завершатся раньше, то они окажутся в состоянии простоя, пока остальные потоки продолжат свою работу. Ранее приведенный пример с простыми числами может плохо выполняться при разбиении на основе диапазонов. Примером, когда такое разбиение оказывается удачным, является вычисление суммы квадратных корней первых 10 миллионов целых чисел:

```
ParallelEnumerable.Range (1, 10000000).Sum (i => Math.Sqrt (i))
```

Метод `ParallelEnumerable.Range` возвращает `ParallelQuery<T>`, поэтому вызывать `AsParallel` впоследствии не придется.

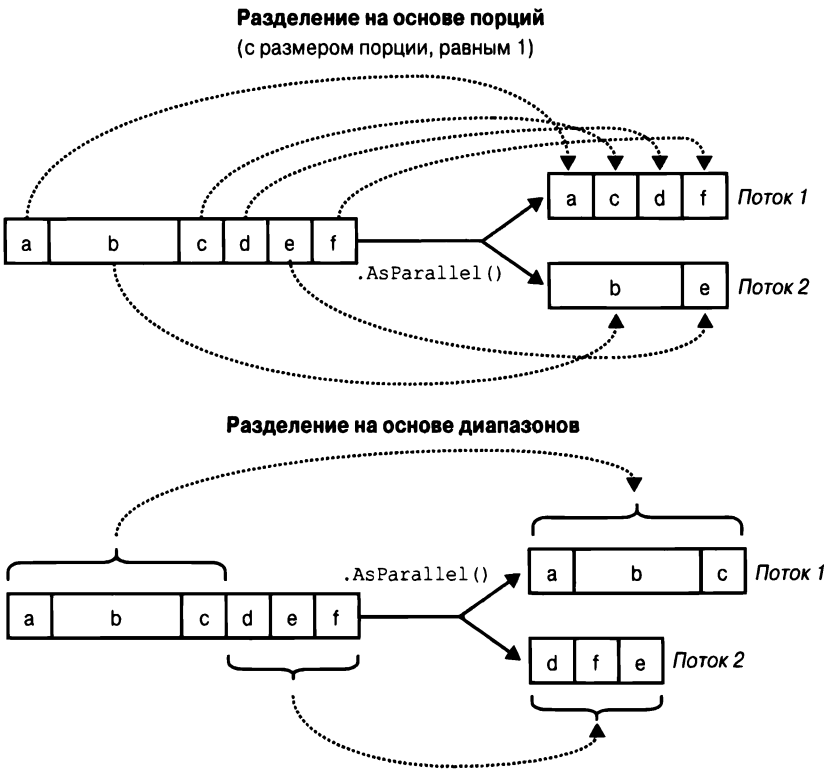



***Рис. 23.4.*** *Сравнение разбиения на основе порций и разбиения на основе диапазонов*

Разбиение на основе диапазонов не обязательно распределяет диапазоны элементов в *смежных* блоках — взамен может быть выбрана "полосовая" стратегия. Например, при наличии двух рабочих потоков один из них может обрабатывать элементы в нечетных позициях, а другой — элементы в четных позициях. Операция `TakeWhile` почти наверняка инициирует полосовую стратегию, чтобы избежать излишней обработки элементов позже в последовательности.

## Оптимизация специального агрегирования

Инфраструктура PLINQ эффективно распараллеливает операции `Sum`, `Average`, `Min` и `Max` без дополнительного вмешательства. Тем не менее, операция `Aggregate` представляет особую трудность для PLINQ. Как было описано в главе 9, операция `Aggregate` выполняет специальное агрегирование. Например, следующий код суммирует последовательность чисел, имитируя операцию `Sum`:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (total, n) => total + n);  // 6
```

В главе 9 также было показано, что для агрегаций *без начального значения* предоставляемый делегат должен быть ассоциативным и коммутативным. Если указанное правило нарушается, тогда инфраструктура PLINQ даст некорректные результаты, поскольку она извлекает *множество начальных значений* из входной последовательности, чтобы выполнять агрегирование нескольких частей последовательности одновременно.

Агрегации с явным начальным значением могут выглядеть как безопасный вариант для PLINQ, но, к сожалению, обычно они выполняются последовательно, т.к. полагаются на единственное начальное значение. Чтобы смягчить данную проблему, PLINQ предлагает еще одну перегруженную версию метода `Aggregate`, которая позволяет указывать множество начальных значений — или скорее *функцию фабрики начальных значений*. В каждом потоке эта функция выполняется для генерации отдельного начального значения, которое фактически становится *локальным для потока* накопителем, куда локально агрегируются элементы.

Потребуется также предоставить функцию для указания способа объединения локального и главного накопителей. Наконец, перегруженная версия метода `Aggregate` (отчасти беспричинно) ожидает делегат для проведения любой финальной трансформации результата (в принципе его легко обеспечить, просто выполняя нужную функцию на результате после его получения). Таким образом, ниже перечислены четыре делегата в порядке их передачи.

- `seedFactory`. Возвращает новый локальный накопитель.
- `updateAccumulatorFunc`. Агрегирует элемент в локальный накопитель.
- `combineAccumulatorFunc`. Объединяет локальный накопитель с главным накопителем.
- `resultSelector`. Применяет любую финальную трансформацию к конечному результату.

В простых сценариях вместо фабрики начальных значений можно указывать просто величину *начального значения*. Такая тактика потерпит неудачу, когда начальное значение относится к ссылочному типу, который требуется изменять, потому что один и тот же экземпляр будет затем совместно использоваться всеми потоками.

В качестве очень простого примера ниже приведен запрос, который суммирует значения в массиве `numbers`:

```
numbers.AsParallel().Aggregate (
 () => 0,                                    // seedFactory
  (localTotal, n) => localTotal + n,         // updateAccumulatorFunc
  (mainTot, localTot) => mainTot + localTot, // combineAccumulatorFunc
  finalResult => finalResult)                // resultSelector
```

Пример несколько надуман, т.к. тот же самый результат можно было бы получить не менее эффективно с применением более простых подходов (скажем, с помощью агрегации без начального значения или, что еще лучше, посредством операции `Sum`). Чтобы предложить более реалистичный пример, предположим, что требуется вычислить частоту появления каждой буквы английского алфавита в заданной строке. Простое последовательное решение может выглядеть следующим образом:

```
string text = "Let's suppose this is a really long string";
var letterFrequencies = new int[26];
foreach (char c in text)
{
  int index = char.ToUpper (c) - 'A';
  if (index >= 0 && index <= 26) letterFrequencies [index]++;
};
```

Примером, когда входной текст может оказаться очень длинным, являются генные цепочки. В таком случае "алфавит" состоит из букв *a*, *c*, *g* и *t*.

Для распараллеливания такого запроса мы могли бы заменить оператор `foreach` вызовом метода `Parallel.ForEach` (как будет показано в следующем разделе), но тогда пришлось бы иметь дело с проблемами параллелизма на разделяемом массиве. Блокирование доступа к данному массиву решило бы проблемы, но уничтожило бы возможность распараллеливания.

Операция `Aggregate` предлагает более аккуратное решение. В этом случае накопителем выступает массив, похожий на массив `letterFrequencies` из предыдущего примера. Ниже представлена последовательная версия, использующая `Aggregate`:

```
int[] result =
  text.Aggregate (
    new int[26],                // Создать "накопитель"
    (letterFrequencies, c) => // Агрегировать букву в этот "накопитель"
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index <= 26) letterFrequencies [index]++;
      return letterFrequencies;
    });
```

А вот параллельная версия, в которой применяется специальная перегруженная версия `Aggregate` из PLINQ:

```
int[] result =
  text.AsParallel().Aggregate (
    () => new int[26],          // Создать новый локальный накопитель
    (localFrequencies, c) =>   // Агрегировать в этот локальный накопитель
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index <= 26) localFrequencies [index]++;
      return localFrequencies;
    },
    // Агрегировать локальный и главный накопители
    (mainFreq, localFreq) =>
      mainFreq.Zip (localFreq, (f1, f2) => f1 + f2).ToArray(),
    finalResult => finalResult // Выполнить любую финальную трансформацию
  );                           // конечного результата
```

Обратите внимание, что функция локального накопителя *изменяет* массив `localFrequencies`. Возможность выполнения такой оптимизации важна — и она законна, поскольку массив `localFrequencies` является локальным для каждого потока.

# Класс `Parallel`

Инфраструктура PFX предоставляет базовую форму структурированного параллелизма через три статических метода в классе `Parallel`.

- `Parallel.Invoke`. Запускает массив делегатов параллельно.
- `Parallel.For`. Выполняет параллельный эквивалент цикла `for` языка C#.
- `Parallel.ForEach`. Выполняет параллельный эквивалент цикла `foreach` языка C#.

Все три метода блокируются вплоть до завершения всей работы. Как и с PLINQ, в случае необработанного исключения оставшиеся рабочие потоки останавливаются после их текущей итерации, а исключение (либо их набор) передается обратно вызывающему потоку внутри оболочки `AggregateException` (как объясняется в разделе "Работа с `AggregateException`" далее в главе).

## Parallel.Invoke

Метод `Parallel.Invoke` запускает массив делегатов `Action` параллельно, после чего ожидает их завершения. Его простейшая версия определена следующим образом:

```
public static void Invoke (params Action[] actions);
```

Как и в PLINQ, методы `Parallel.*` оптимизированы для выполнения работы с интенсивными вычислениями, но не интенсивным вводом-выводом. Тем не менее, загрузка двух веб-страниц за раз позволяет легко продемонстрировать использование метода `Parallel.Invoke`:

```
Parallel.Invoke (
  () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),
  () => new WebClient().DownloadFile ("http://www.jaoo.dk", "jaoo.html"));
```

На первый взгляд код выглядит удобным сокращением для создания и ожидания двух привязанных к потокам объектов `Task`. Однако существует важное отличие: метод `Parallel.Invoke` будет работать по-прежнему эффективно, даже если ему передать массив из миллиона делегатов. Причина в том, что он *разбивает* большое количество элементов на пакеты, которые назначает небольшому числу существующих объектов `Task`, а не создает отдельный объект `Task` для каждого делегата.

Как и со всеми методами класса `Parallel`, объединение результатов возлагается полностью на вас. Это значит, что вы должны помнить о безопасности в отношении потоков. Например, приведенный ниже код не является безопасным к потокам:

```
var data = new List<string>();
Parallel.Invoke (
  () => data.Add (new WebClient().DownloadString ("http://www.foo.com")),
  () => data.Add (new WebClient().DownloadString ("http://www.far.com")));
```

Помещение кода добавления в список внутрь блокировки решило бы проблему, но блокировка создаст узкое место в случае гораздо более крупных массивов быстро выполняющихся делегатов. Лучшее решение предусматривает использование безопасных к потокам коллекций, которые рассматриваются далее в главе — идеальным вариантом в данном случае была бы коллекция `ConcurrentBag`.

Метод `Parallel.Invoke` также имеет перегруженную версию, принимающую объект `ParallelOptions`:

```
public static void Invoke (ParallelOptions options,
                           params Action[] actions);
```

С помощью объекта `ParallelOptions` можно вставить признак отмены, ограничить максимальную степень параллелизма и указать специальный планировщик задач. Признак отмены играет важную роль, когда выполняется (приблизительно) большее количество задач, чем есть процессорных ядер: при отмене все незапущенные делегаты будут отброшены. Однако любые уже выполняющиеся делегаты продолжат свою работу вплоть до ее завершения. В разделе "Отмена" ранее в главе приводился пример применения признаков отмены.

## Parallel.For и Parallel.ForEach

Методы Parallel.For и Parallel.ForEach реализуют эквиваленты циклов for и foreach из C#, но с выполнением каждой итерации параллельно, а не последовательно. Ниже показаны их (простейшие) сигнатуры:

```
public static ParallelLoopResult For (
  int fromInclusive, int toExclusive, Action<int> body)
public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource> body)
```

Следующий последовательный цикл for:

```
for (int i = 0; i < 100; i++)
  Foo (i);
```

распараллеливается примерно так:

```
Parallel.For (0, 100, i => Foo (i));
```

или еще проще:

```
Parallel.For (0, 100, Foo);
```

А представленный далее последовательный цикл foreach:

```
foreach (char c in "Hello, world")
  Foo (c);
```

распараллеливается следующим образом:

```
Parallel.ForEach ("Hello, world", Foo);
```

Рассмотрим практический пример. Если мы импортируем пространство имен System.Security.Cryptography, то сможем генерировать шесть строк с парами открытого и секретного ключей параллельно:

```
var keyPairs = new string[6];
Parallel.For (0, keyPairs.Length,
              i => keyPairs[i] = RSA.Create().ToXmlString (true));
```

Как и в случае Parallel.Invoke, методам Parallel.For и Parallel.ForEach можно передавать большое количество элементов работы и они будут эффективно распределены по нескольким задачам.

Последний запрос можно также построить с помощью PLINQ:

```
string[] keyPairs =
  ParallelEnumerable.Range (0, 6)
  .Select (i => RSA.Create().ToXmlString (true))
  .ToArray();
```

### Сравнение внешних и внутренних циклов

Методы Parallel.For и Parallel.ForEach обычно лучше всего работают на внешних, а не на внутренних циклах. Причина в том, что посредством внешних циклов вы предлагаете распараллеливать более крупные порции работы, снижая накладные расходы по управлению. Распараллеливание сразу внутренних и внешних циклов обычно излишне. В следующем примере для получения ощутимой выгоды от распараллеливания внутреннего цикла обычно требуется более 100 ядер:

```
Parallel.For (0, 100, i =>
{
  Parallel.For (0, 50, j => Foo (i, j)); // Внутренний цикл лучше
});                                      // выполнять последовательно.
```

### Индексированная версия Parallel.ForEach

Временами полезно знать индекс итерации цикла. В случае последовательного цикла foreach это легко:

```
int i = 0;
foreach (char c in "Hello, world")
  Console.WriteLine (c.ToString() + i++);
```

Однако инкрементирование разделяемой переменной не является безопасным к потокам в параллельном контексте. Взамен должна использоваться следующая версия ForEach:

```
public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource,ParallelLoopState,long> body)
```

Мы проигнорируем класс ParallelLoopState (он будет рассматриваться в следующем разделе). Пока что нас интересует третий параметр типа long, который отражает индекс цикла:

```
Parallel.ForEach ("Hello, world", (c, state, i) =>
{
  Console.WriteLine (c.ToString() + i);
});
```

Чтобы применить такой прием в практическом примере, вернемся к программе проверки орфографии, которую мы писали с помощью PLINQ. Следующий код загружает словарь и массив с миллионом слов для целей тестирования:

```
if (!File.Exists ("WordLookup.txt"))   // Содержит около 150 000 слов
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);

var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";         // Внесение пары
wordsToTest [23456] = "wubsie";         // орфографических ошибок.
```

Мы можем выполнить проверку орфографии в массиве wordsToTest с использованием индексированной версии Parallel.ForEach:

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

Обратите внимание, что мы должны объединять результаты в безопасную к потокам коллекцию: по сравнению с применением PLINQ необходимость в таком действии считается недостатком. Преимущество перед PLINQ связано с тем, что мы избегаем использования индексированной операции запроса Select, которая менее эффективна, чем индексированная версия метода ForEach.

### ParallelLoopState: раннее прекращение циклов

Поскольку тело цикла в параллельном методе For или ForEach представляет собой делегат, выйти из цикла до его полного завершения с помощью оператора break не получится. Вместо этого придется вызвать метод Break или Stop на объекте ParallelLoopState:

```
public class ParallelLoopState
{
  public void Break();
  public void Stop();

  public bool IsExceptional { get; }
  public bool IsStopped { get; }
  public long? LowestBreakIteration { get; }
  public bool ShouldExitCurrentIteration { get; }
}
```

Получить объект ParallelLoopState довольно просто: все версии методов For и ForEach перегружены для приема тела цикла типа Action<TSource, ParallelLoop State>. Таким образом, для распараллеливания следующего цикла:

```
foreach (char c in "Hello, world")
  if (c == ',')
    break;
  else
    Console.Write (c);
```

нужно поступить так:

```
Parallel.ForEach ("Hello, world", (c, loopState) =>
{
  if (c == ',')
    loopState.Break();
  else
    Console.Write (c);
});

// ВЫВОД: Hlloe
```

В выводе несложно заметить, что тела циклов могут завершаться в произвольном порядке. Помимо такого отличия вызов Break выдает, *по меньшей мере*, те же самые элементы, что и при последовательном выполнении цикла: приведенный пример будет всегда выводить *минимум* буквы *H*, *e*, *l*, *l* и *o* в каком-нибудь порядке. Напротив, вызов Stop вместо Break приводит к принудительному завершению всех потоков сразу после их текущей итерации. В данном примере вызов Stop может дать подмножество букв *H*, *e*, *l*, *l* и *o*, если другой поток отстал. Вызов Stop полезен, когда обнаружено то, что требовалось найти, или выяснилось, что что-то пошло не так, и потому результаты просматриваться не будут.

Методы Parallel.For и Parallel.ForEach возвращают объект ParallelLoopResult, который открывает доступ к свойствам с именами IsCompleted и LowestBreakIteration. Они сообщают, полностью ли завершился цикл, и если это не так, то на какой итерации он был прекращен. Если свойство LowestBreakIteration возвращает null, тогда в цикле вызывался метод Stop (а не Break).

В случае длинного тела цикла может потребоваться прервать другие потоки где-то на полпути тела метода при раннем вызове `Break` или `Stop`, что реализуется за счет опроса свойства `ShouldExitCurrentIteration` в различных местах кода. Указанное свойство принимает значение `true` немедленно после вызова `Stop` или очень скоро после вызова `Break`.

Свойство `ShouldExitCurrentIteration` также становится равным `true` после запроса отмены либо в случае генерации исключения в цикле.

Свойство `IsExceptional` позволяет узнать, произошло ли исключение в другом потоке. Любое необработанное исключение приведет к останову цикла после текущей итерации каждого потока: чтобы избежать его, вы должны явно обрабатывать исключения в своем коде.

### Оптимизация посредством локальных значений

Методы `Parallel.For` и `Parallel.ForEach` предлагают набор перегруженных версий, которые работают с аргументом обобщенного типа по имени `TLocal`. Такие перегруженные версии призваны помочь оптимизировать объединение данных из циклов с интенсивными итерациями. Простейшая из них выглядит следующим образом:

```
public static ParallelLoopResult For <TLocal> (
  int fromInclusive,
  int toExclusive,
  Func <TLocal> localInit,
  Func <int, ParallelLoopState, TLocal, TLocal> body,
  Action <TLocal> localFinally);
```

На практике данные методы редко востребованы, т.к. их целевые сценарии в основном покрываются PLINQ (что, в принципе, хорошо, поскольку иногда их перегруженные версии выглядят слегка устрашающими).

По существу вот в чем заключается проблема: предположим, что необходимо просуммировать квадратные корни чисел от 1 до 10 000 000. Вычисление 10 миллионов квадратных корней легко распараллеливается, но суммирование их значений — дело хлопотное, т.к. обновление итоговой суммы должно быть помещено внутрь блокировки:

```
object locker = new object();
double total = 0;
Parallel.For (1, 10000000,
              i => { lock (locker) total += Math.Sqrt (i); });
```

Выигрыш от распараллеливания более чем нивелируется ценой получения 10 миллионов блокировок, а также блокированием результата.

Однако в реальности нам *не нужны* 10 миллионов блокировок. Представьте себе команду волонтеров по сборке большого объема мусора. Если все работники совместно пользуются единственным мусорным ведром, то хождения к нему и состязания сделают процесс крайне неэффективным. Очевидное решение предусматривает снабжение каждого работника собственным или "локальным" мусорным ведром, которое время от времени опустошается в главный контейнер.

Именно таким способом работают версии `TLocal` методов `For` и `ForEach`. Волонтеры — это внутренние рабочие потоки, а *локальное значение* представляет локальное мусорное ведро.

Чтобы класс Parallel справился с этой работой, необходимо предоставить два дополнительных делегата.

1. Делегат, который указывает, каким образом инициализировать новое локальное значение.
2. Делегат, который указывает, каким образом объединять локальную агрегацию с главным значением.

Кроме того, вместо возвращения void делегат тела цикла должен возвращать новую агрегацию для локального значения. Ниже приведен переделанный пример:

```
object locker = new object();
double grandTotal = 0;
Parallel.For (1, 10000000,
  () => 0.0,                      // Инициализировать локальное значение.
  (i, state, localTotal) =>       // Делегат тела цикла. Обратите внимание,
    localTotal + Math.Sqrt (i),   // что он возвращает новый локальный итог.
  localTotal =>                                  // Добавить локальное значение
    { lock (locker) grandTotal += localTotal; }  // к главному значению.
);
```

Здесь по-прежнему требуется блокировка, но только вокруг агрегирования локального значения с общей суммой, что делает процесс гораздо эффективнее.

Как утверждалось ранее, PLINQ часто хорошо подходит для таких сценариев. Распараллелить наш пример с помощью PLINQ можно было бы просто так:

```
ParallelEnumerable.Range (1, 10000000)
                  .Sum (i => Math.Sqrt (i))
```

(Обратите внимание, что мы применяем ParallelEnumerable для обеспечения *разбиения на основе диапазонов*: в данном случае оно улучшает производительность, потому что все числа требуют равного времени на обработку.)

В более сложных сценариях вместо Sum может использоваться LINQ-операция Aggregate. Если вы предоставите фабрику локальных начальных значений, то ситуация будет в чем-то аналогична предоставлению функции локальных значений для Parallel.For.

# Параллелизм задач

*Параллелизм задач* – это подход самого низкого уровня к распараллеливанию с применением инфраструктуры PFX. Классы для работы на таком уровне определены в пространстве имен System.Threading.Tasks и включают перечисленные ниже.

| Класс | Назначение |
|---|---|
| Task | Для управления единицей работы |
| Task<TResult> | Для управления единицей работы с возвращаемым значением |
| TaskFactory | Для создания задач |
| TaskFactory<TResult> | Для создания задач и продолжений с тем же самым возвращаемым типом |
| TaskScheduler | Для управления планированием задач |
| TaskCompletionSource | Для ручного управления рабочим потоком действий задачи |

Основы задач были раскрыты в главе 14; в настоящем разделе мы рассмотрим расширенные возможности задач, которые ориентированы на параллельное программирование. В частности, будут обсуждаться следующие темы:

- тонкая настройка планирования задачи;
- установка отношения "родительская/дочерняя", когда одна задача запускается из другой;
- расширенное использование продолжений;
- класс `TaskFactory`.

Библиотека параллельных задач (TPL) позволяет создавать сотни (или даже тысячи) задач с минимальными накладными расходами. Но если необходимо создавать миллионы задач, то для поддержания эффективности эти задачи понадобится организовывать в более крупные единицы работы. Класс `Parallel` и PLINQ делают автоматически.

В Visual Studio предусмотрено окно для мониторинга задач (Debug⇨Window⇨Parallel Tasks (Отладка⇨Окно⇨Параллельные задачи)). Окно Parallel Tasks (Параллельные задачи) эквивалентно окну Threads (Потоки), но предназначено для задач. Окно Parallel Stacks (Параллельные стеки) также поддерживает специальный режим для задач.

## Создание и запуск задач

Как было описано в главе 14, метод `Task.Run` создает и запускает объект `Task` или `Task<TResult>`. На самом деле `Task.Run` является сокращением для вызова метода `Task.Factory.StartNew`, который предлагает более высокую гибкость через дополнительные перегруженные версии.

### Указание объекта состояния

Метод `Task.Factory.StartNew` позволяет указывать объект *состояния*, который передается целевому методу. Сигнатура целевого метода должна в данном случае содержать одиночный параметр типа `object`:

```
static void Main()
{
  var task = Task.Factory.StartNew (Greet, "Hello");
  task.Wait();  // Ожидать, пока задача завершится.
}
static void Greet (object state) { Console.Write (state); }  // Hello
```

Такой прием дает возможность избежать затрат на замыкание, требуемое для выполнения лямбда-выражения, которое вызывает метод `Greet`. Это является микро-оптимизацией и редко необходимо на практике, так что мы можем оставить объект состояния для более полезного сценария — назначение задаче значащего имени. Затем для запрашивания имени можно применять свойство `AsyncState`:

```
static void Main()
{
  var task = Task.Factory.StartNew (state => Greet ("Hello"), "Greeting");
  Console.WriteLine (task.AsyncState);  // Greeting
  task.Wait();
}
static void Greet (string message) { Console.Write (message); }
```

Среда Visual Studio отображает значение свойства `AsyncState` каждой задачи в окне Parallel Tasks, так что значащее имя задачи может основательно упростить отладку.

### TaskCreationOptions

Настроить выполнение задачи можно за счет указания перечисления `TaskCreationOptions` при вызове `StartNew` (или создании объекта `Task`). `TaskCreationOptions` — перечисление флагов со следующими (комбинируемыми) значениями:

```
LongRunning, PreferFairness, AttachedToParent
```

Значение `LongRunning` предлагает планировщику выделить для задачи поток; как было показано в главе 14, это полезно для задач с интенсивным вводом-выводом, а также для длительно выполняющихся задач, которые иначе могут заставить кратко выполняющиеся задачи ожидать чрезмерно долгое время перед тем, как они будут запланированы.

Значение `PreferFairness` сообщает планировщику о необходимости попытаться обеспечить планирование задач в том порядке, в каком они были запущены. Обычно планировщик может поступать иначе, потому что он внутренне оптимизирует планирование задач с использованием локальных очередей захвата работ — оптимизация, позволяющая создавать *дочерние* задачи без накладных расходов на состязания, которые в противном случае возникли бы при доступе к единственной очереди работ. Дочерняя задача создается путем указания значения `AttachedToParent`.

### Дочерние задачи

Когда одна задача запускает другую, можно дополнительно установить отношение "родительская/дочерняя":

```
Task parent = Task.Factory.StartNew (() =>
{
  Console.WriteLine ("I am a parent");
  Task.Factory.StartNew (() =>     // Отсоединенная задача
  {
    Console.WriteLine ("I am detached");
  });
  Task.Factory.StartNew (() =>     // Дочерняя задача
  {
    Console.WriteLine ("I am a child");
  }, TaskCreationOptions.AttachedToParent);
});
```

Дочерняя задача специфична тем, что при ожидании завершения *родительской* задачи ожидаются также и любые ее дочерние задачи. До этой точки поднимаются любые дочерние исключения:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
var parent = Task.Factory.StartNew (() =>
{
  Task.Factory.StartNew (() =>  // Дочерняя
  {
    Task.Factory.StartNew (() => { throw null; }, atp);  // Внучатая
  }, atp);
});
```

```
// Следующий вызов генерирует исключение NullReferenceException
// (помещенное в оболочку AggregateExceptions):
parent.Wait();
```

Как вскоре будет показано, прием может быть особенно полезным, когда дочерняя задача является продолжением.

## Ожидание на множестве задач

В главе 14 упоминалось, что организовать ожидание на одиночной задаче можно либо вызовом ее метода `Wait`, либо обращением к ее свойству `Result` (в случае `Task<TResult>`). Можно также реализовать ожидание сразу на множестве задач — с помощью статических методов `Task.WaitAll` (ожидание завершения всех указанных задач) и `Task.WaitAny` (ожидание завершения какой-то одной задачи).

Метод `WaitAll` похож на ожидание каждой задачи по очереди, но более эффективен тем, что требует (максимум) одного переключения контекста. Кроме того, если одна или более задач генерируют необработанное исключение, то `WaitAll` по-прежнему ожидает каждую задачу и затем генерирует исключение `AggregateException`, в котором накоплены исключения из всех отказавших задач (ситуация, когда класс `AggregateException` по-настоящему полезен). Ниже показан эквивалентный код:

```
// Предполагается, что t1, t2 и t3 - задачи:
var exceptions = new List<Exception>();
try { t1.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t2.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t3.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
if (exceptions.Count > 0) throw new AggregateException (exceptions);
```

Вызов `WaitAny` эквивалентен ожиданию события `ManualResetEventSlim`, которое сигнализируется каждой задачей, как только она завершена.

Помимо времени тайм-аута методам `Wait` можно также передавать *признак отмены*: это позволяет отменить ожидание, *но не саму задачу*.

## Отмена задач

При запуске задачи можно дополнительно передавать признак отмены. Если позже через данный признак произойдет отмена, то задача войдет в состояние `Canceled` (отменена):

```
var cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
cts.CancelAfter (500);

Task task = Task.Factory.StartNew (() =>
{
  Thread.Sleep (1000);
  token.ThrowIfCancellationRequested();  // Проверить запрос отмены
}, token);

try { task.Wait(); }
catch (AggregateException ex)
{
  Console.WriteLine (ex.InnerException is TaskCanceledException); // True
  Console.WriteLine (task.IsCanceled);                            // True
  Console.WriteLine (task.Status);                                //Canceled
}
```

TaskCanceledException — подкласс класса OperationCanceledException. Если нужно явно сгенерировать исключение OperationCanceledException (вместо вызова token.ThrowIfCancellationRequested), тогда потребуется передать признак отмены конструктору OperationCanceledException. Если это не сделано, то задача не войдет в состояние TaskStatus.Canceled и не будет инициировать продолжение OnlyOnCanceled.

Если задача отменяется еще до своего запуска, тогда она не будет запланирована — в таком случае исключение OperationCanceledException сгенерируется немедленно.

Поскольку признаки отмены распознаются другими API-интерфейсами, их можно передавать другим конструкциям и отмена будет распространяться гладким образом:

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;
Task task = Task.Factory.StartNew (() =>
{
  // Передать признак отмены в запрос PLINQ:
  var query = someSequence.AsParallel().WithCancellation (token)...
  ...перечислить результаты запроса...
});
```

Вызов Cancel на cancelSource в рассмотренном примере приведет к отмене запроса PLINQ с генерацией исключения OperationCanceledException в теле задачи, которое затем отменит задачу.

Признаки отмены, которые можно передавать в методы, подобные Wait и CancelAndWait, позволяют отменить операцию *ожидания*, а не саму задачу.

## Продолжение

Метод ContinueWith выполняет делегат сразу после завершения задачи:

```
Task task1 = Task.Factory.StartNew (() => Console.Write ("antecedant.."));
Task task2 = task1.ContinueWith (ant => Console.Write ("..continuation"));
```

Как только задача task1 (*предшественник*) завершается, отказывает или отменяется, запускается задача task2 (*продолжение*). (Если задача task1 была завершена до того, как выполнилась вторая строка кода, то задача task2 будет запланирована для выполнения немедленно.) Аргумент ant, переданный лямбда-выражению продолжения, представляет собой ссылку на предшествующую задачу. Сам метод ContinueWith возвращает задачу, облегчая добавление дополнительных продолжений.

По умолчанию предшествующая задача и задача продолжения могут выполняться в разных потоках. Указав TaskContinuationOptions.ExecuteSynchronously при вызове ContinueWith, можно заставить их выполняться в одном и том же потоке: это позволяет улучшить производительность при мелкомодульных продолжениях за счет уменьшения косвенности.

### Продолжение и Task<TResult>

Подобно обычным задачам продолжения могут иметь тип Task<TResult> и возвращать данные. В следующем примере мы вычисляем Math.Sqrt(8*2) с применением последовательности соединенных в цепочку задач и выводим результат:

```
Task.Factory.StartNew<int> (() => 8)
  .ContinueWith (ant => ant.Result * 2)
  .ContinueWith (ant => Math.Sqrt (ant.Result))
  .ContinueWith (ant => Console.WriteLine (ant.Result)); // 4
```

Из-за стремления к простоте этот пример получился несколько неестественным; в реальности такие лямбда-выражения могли бы вызывать функции с интенсивными вычислениями.

### Продолжение и исключения

Продолжение может узнать, отказал ли предшественник, запросив свойство `Exception` предшествующей задачи или просто вызвав метод `Result/Wait` и перехватив результирующее исключение `AggregateException`. Если предшественник отказал, и то же самое сделало продолжение, тогда исключение считается *необнаруженным*; в таком случае при последующей обработке задачи сборщиком мусора будет инициировано статическое событие `TaskScheduler.UnobservedTaskException`.

Безопасный шаблон предполагает повторную генерацию исключений предшественника. До тех пор пока ожидается продолжение, исключение будет распространяться и повторно генерироваться для ожидающей задачи:

```
Task continuation = Task.Factory.StartNew    (()  => { throw null; })
                                  .ContinueWith (ant =>
  {
    ant.Wait();
    // Продолжить обработку...
  });
continuation.Wait(); // Исключение теперь передается обратно вызывающей задаче.
```

Другой способ иметь дело с исключениями предусматривает указание разных продолжений для исходов с и без исключений, что делается с помощью перечисления `TaskContinuationOptions`:

```
Task task1 = Task.Factory.StartNew (() => { throw null; });
Task error = task1.ContinueWith (ant => Console.Write (ant.Exception),
                                 TaskContinuationOptions.OnlyOnFaulted);
Task ok = task1.ContinueWith (ant => Console.Write ("Success!"),
                              TaskContinuationOptions.NotOnFaulted);
```

Как вскоре будет показано, такой шаблон особенно удобен в сочетании с дочерними задачами.

Следующий расширяющий метод "поглощает" необработанные исключения задачи:

```
public static void IgnoreExceptions (this Task task)
{
  task.ContinueWith (t => { var ignore = t.Exception; },
    TaskContinuationOptions.OnlyOnFaulted);
}
```

(Метод может быть улучшен добавлением кода для регистрации исключения.) Вот как его можно использовать:

```
Task.Factory.StartNew (() => { throw null; }).IgnoreExceptions();
```

### Продолжение и дочерние задачи

Мощная особенность продолжений связана с тем, что они запускаются, только когда завершены все дочерние задачи (рис. 23.5). В этой точке любые исключения, сгенерированные дочерними задачами, маршализируются в продолжение.

В следующем примере мы начинаем три дочерние задачи, каждая из которых генерирует исключение `NullReferenceException`.

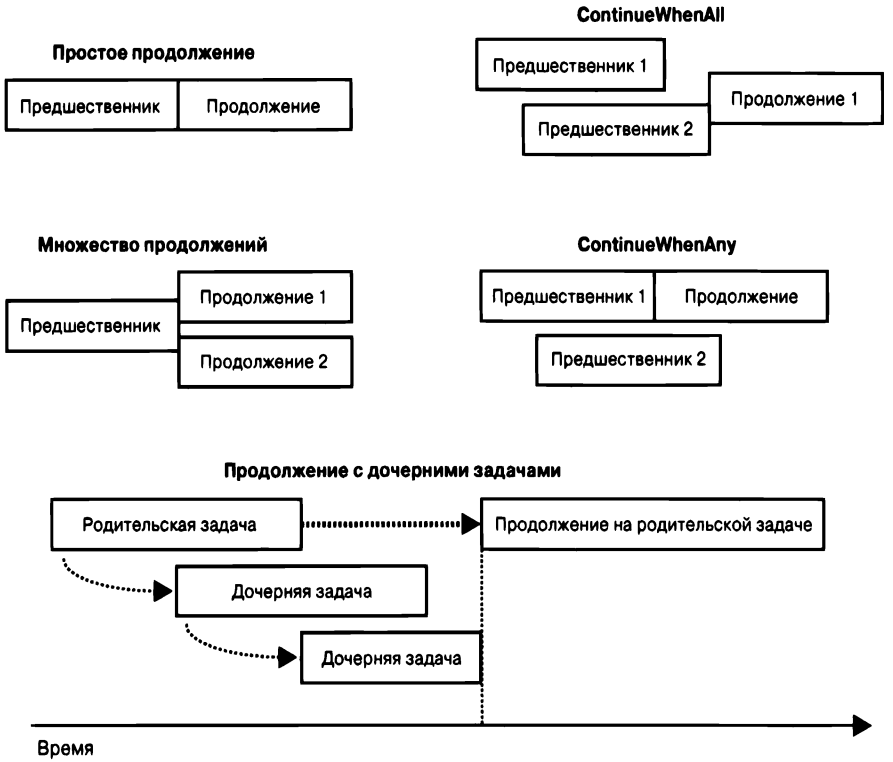



*Рис. 23.5. Продолжения*

Затем мы перехватываем все исключения сразу через продолжение на родительской задаче:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
Task.Factory.StartNew (() =>
{
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
})
.ContinueWith (p => Console.WriteLine (p.Exception),
                    TaskContinuationOptions.OnlyOnFaulted);
```

## Условные продолжения

По умолчанию продолжение планируется *безусловным образом* – независимо от того, завершена предшествующая задача, сгенерировано исключение или задача была отменена. Такое поведение можно изменить с помощью набора (комбинируемых) флагов, определенных в перечислении `TaskContinuationOptions`. Вот три основных флага, которые управляют условным продолжением:

```
NotOnRanToCompletion = 0x10000,
NotOnFaulted = 0x20000,
NotOnCanceled = 0x40000,
```

Флаги являются субтрактивными в том смысле, что чем больше их применяется, тем менее вероятно выполнение продолжения. Для удобства также предоставляются следующие заранее скомбинированные значения:

```
OnlyOnRanToCompletion = NotOnFaulted  | NotOnCanceled,
OnlyOnFaulted = NotOnRanToCompletion  | NotOnCanceled,
OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted
```

(Объединение всех флагов `Not*` (`NotOnRanToCompletion`, `NotOnFaulted`, `NotOnCanceled`) бессмысленно, т.к. в результате продолжение будет всегда отменяться.)

Наличие "RanToCompletion" в имени означает успешное завершение предшествующей задачи – без отмены или необработанных исключений.

Наличие "Faulted" в имени означает, что в предшествующей задаче было сгенерировано необработанное исключение.

Наличие "Canceled" в имени означает одну из следующих двух ситуаций.

- Предшествующая задача была отменена через ее признак отмены. Другими словами, в предшествующей задаче было сгенерировано исключение `OperationCanceledException`, свойство `CancellationToken` которого соответствует тому, что было передано предшествующей задаче во время запуска.
- Предшествующая задача была неявно отменена, поскольку она не удовлетворила предикат условного продолжения.

Важно понимать, что когда продолжение не выполнилось из-за упомянутых флагов, оно не забыто и не отброшено – это продолжение *отменено*. Другими словами, любые продолжения на самом отмененном продолжении *затем запустятся* – если только вы не указали в условии флаг `NotOnCanceled`. Например, взгляните на приведенный далее код:

```
Task t1 = Task.Factory.StartNew (...);
Task fault = t1.ContinueWith (ant => Console.WriteLine ("fault"),
                                     TaskContinuationOptions.OnlyOnFaulted);
Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"));
```

Несложно заметить, что задача `t3` всегда будет запланирована – даже если `t1` не генерирует исключение (рис. 23.6). Причина в том, что если задача `t1` завершена успешно, тогда задача `fault` будет *отменена*, и с учетом отсутствия ограничений продолжения задача `t3` будет запущена безусловным образом.

Если нужно, чтобы задача `t3` выполнялась, только если действительно была запущена задача `fault`, то потребуется поступить так:

```
Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"),
                                     TaskContinuationOptions.NotOnCanceled);
```

(В качестве альтернативы мы могли бы указать `OnlyOnRanToCompletion`; разница в том, что тогда задача `t3` не запустилась бы в случае генерации исключения внутри задачи `fault`.)

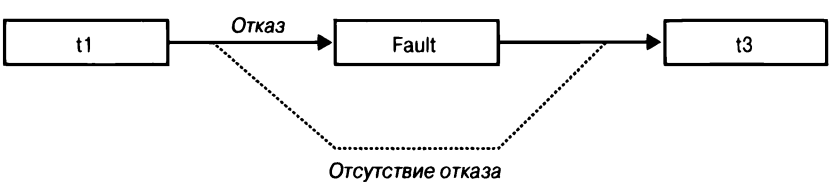



***Рис. 23.6.** Условные продолжения*

### Продолжение на основе множества предшествующих задач

С помощью методов ContinueWhenAll и ContinueWhenAny класса TaskFactory выполнение продолжения можно планировать на основе завершения множества предшествующих задач. Однако эти методы стали избыточными после появления комбинаторов задач, которые обсуждались в главе 14 (WhenAll и WhenAny). В частности, при наличии следующих задач:

```
var task1 = Task.Run (() => Console.Write ("X"));
var task2 = Task.Run (() => Console.Write ("Y"));
```

вот как можно запланировать выполнение продолжения, когда обе они завершатся:

```
var continuation = Task.Factory.ContinueWhenAll (
  new[] { task1, task2 }, tasks => Console.WriteLine ("Done"));
```

Тот же результат легко получить с помощью комбинатора задач WhenAll:

```
var continuation = Task.WhenAll (task1, task2)
                       .ContinueWith (ant => Console.WriteLine ("Done"));
```

### Множество продолжений на единственной предшествующей задаче

Вызов ContinueWith более одного раза на той же самой задаче создает множество продолжений на единственном предшественнике. Когда предшественник завершается, все продолжения запускаются вместе (если только не было указано значение TaskContinuationOptions.ExecuteSynchronously, из-за чего продолжения будут выполняться последовательно). Следующий код ожидает одну секунду, а затем выводит на консоль либо XY, либо YX:

```
var t = Task.Factory.StartNew (() => Thread.Sleep (1000));
t.ContinueWith (ant => Console.Write ("X"));
t.ContinueWith (ant => Console.Write ("Y"));
```

## Планировщики задач

*Планировщик задач* распределяет задачи по потокам и представлен абстрактным классом TaskScheduler. В .NET Framework предлагаются две конкретные реализации: *стандартный планировщик*, который работает в тандеме с пулом потоков CLR, и *планировщик контекста синхронизации*. Последний предназначен (главным образом) для содействия в работе с потоковой моделью WPF и Windows Forms, которая требует, чтобы доступ к элементам управления пользовательского интерфейса осуществлялся только из создавшего их потока (см. раздел "Многопоточность в обогащенных клиентских приложениях" в главе 14). Захватив такой планировщик, мы можем сообщить задаче или продолжению о выполнении в этом контексте:

```
// Предположим, что мы находимся в потоке пользовательского
// интерфейса внутри приложения Windows Forms или WPF:
_uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
```

Предполагая, что Foo — метод с интенсивными вычислениями, возвращающий строку, а lblResult — метка WPF или Windows Forms, вот как можно было бы безопасно обновить метку после завершения операции:

```
Task.Run (() => Foo())
  .ContinueWith (ant => lblResult.Content = ant.Result, _uiScheduler);
```

Разумеется, для действий подобного рода чаще будут использоваться асинхронные функции C#.

Возможно также написание собственного планировщика задач (путем создания подкласса TaskScheduler), хотя это делается только в очень специализированных сценариях. Для специального планирования чаще всего будет применяться класс TaskCompletionSource.

### TaskFactory

Когда вызывается Task.Factory, происходит обращение к статическому свойству класса Task, которое возвращает стандартный объект фабрики задач, т.е. TaskFactory. Назначение фабрики задач заключается в создании задач – в частности, трех их видов:

- "обычных" задач (через метод StartNew);
- продолжений с множеством предшественников (через методы ContinueWhenAll и ContinueWhenAny);
- задач, которые являются оболочками для методов, следующих устаревшему шаблону APM (через метод FromAsync; см. раздел "Устаревшие шаблоны" в главе 14).

Еще один способ создания задач предусматривает создание экземпляра Task и вызов метода Start. Тем не менее, подобным образом можно создавать только "обычные" задачи, но не продолжения.

#### Создание собственных фабрик задач

Класс TaskFactory – не *абстрактная* фабрика: на самом деле вы можете создавать объекты данного класса, что удобно, когда нужно многократно создавать задачи с использованием тех же самых (нестандартных) значений для TaskCreationOptions, TaskContinuationOptions или TaskScheduler. Например, если требуется многократно создавать длительно выполняющиеся родительские задачи, тогда специальную фабрику можно построить следующим образом:

```
var factory = new TaskFactory (
  TaskCreationOptions.LongRunning | TaskCreationOptions.AttachedToParent,
  TaskContinuationOptions.None);
```

Затем создание задач сводится просто к вызову метода StartNew на фабрике:

```
Task task1 = factory.StartNew (Method1);
Task task2 = factory.StartNew (Method2);
...
```

Специальные опции продолжения применяются в случае вызова методов ContinueWhenAll и ContinueWhenAny.

## Работа с AggregateException

Как вы уже видели, инфраструктура PLINQ, класс Parallel и объекты Task автоматически маршализируют исключения потребителю. Чтобы понять, почему это важно, рассмотрим показанный ниже запрос LINQ, который на первой итерации генерирует исключение DivideByZeroException:

```
try
{
  var query = from i in Enumerable.Range (0, 1000000)
              select 100 / i;
  ...
}
```

```
catch (DivideByZeroException)
{
  ...
}
```

Если запросить у инфраструктуры PLINQ распараллеливание такого запроса, и она проигнорирует обработку исключений, то вполне возможно, что исключение `DivideByZeroException` сгенерируется в *отдельном потоке*, пропустив ваш блок `catch` и вызвав аварийное завершение приложения.

Поэтому исключения автоматически перехватываются и повторно генерируются для вызывающего потока. Но, к сожалению, дело не сводится просто к перехвату `DivideByZeroException`. Поскольку параллельные библиотеки задействуют множество потоков, вполне возможна одновременная генерация двух и более исключений. Чтобы обеспечить получение сведений обо всех исключениях, по указанной причине исключения помещаются в контейнер `AggregateException`, свойство `InnerExceptions` которого содержит каждое из перехваченных исключений:

```
try
{
  var query = from i in ParallelEnumerable.Range (0, 1000000)
              select 100 / i;
  // Выполнить перечисление результатов запроса
  ...
}
catch (AggregateException aex)
{
  foreach (Exception ex in aex.InnerExceptions)
    Console.WriteLine (ex.Message);
}
```

Как инфраструктура PLINQ, так и класс `Parallel` при обнаружении первого исключения заканчивают выполнение запроса или цикла, не обрабатывая любые последующие элементы либо итерации тела цикла. Однако до завершения текущей итерации цикла могут быть сгенерированы дополнительные исключения. Первое возникшее исключение в `AggregateException` доступно через свойство `InnerException`.

## Flatten и Handle

Класс `AggregateException` предоставляет пару методов для упрощения обработки исключений: `Flatten` и `Handle`.

### Flatten

Объекты `AggregateException` довольно часто будут содержать другие объекты `AggregateException`. Пример, когда подобное может произойти – ситуация, при которой дочерняя задача генерирует исключение. Чтобы упростить обработку, можно устранить любой уровень вложения, вызвав метод `Flatten`. Этот метод возвращает новый объект `AggregateException` с обычным плоским списком внутренних исключений:

```
catch (AggregateException aex)
{
  foreach (Exception ex in aex.Flatten().InnerExceptions)
    myLogWriter.LogException (ex);
}
```

### Handle

Иногда полезно перехватывать исключения только специфических типов, а исключения других типов генерировать повторно.

Метод `Handle` класса `AggregateException` предлагает удобное сокращение. Он принимает предикат исключений, который будет запускаться на каждом внутреннем исключении:

```
public void Handle (Func<Exception, bool> predicate)
```

Если предикат возвращает `true`, то считается, что исключение "обработано". После того, как делегат запустится на всех исключениях, произойдет следующее:

- если все исключения были "обработаны" (делегат возвратил `true`), то исключение не генерируется повторно;
- если были исключения, для которых делегат возвратил `false` ("необработанные"), то строится новый объект `AggregateException`, содержащий такие исключения, и затем он генерируется повторно.

Например, приведенный далее код в конечном итоге повторно генерирует другой объект `AggregateException`, который содержит одиночное исключение `NullReferenceException`:

```
var parent = Task.Factory.StartNew (() =>
{
  // Мы сгенерируем 3 исключения сразу, используя 3 дочерние задачи:
  int[] numbers = { 0 };
  var childFactory = new TaskFactory
   (TaskCreationOptions.AttachedToParent, TaskContinuationOptions.None);
  childFactory.StartNew (() => 5 / numbers[0]);  // Деление на ноль
  childFactory.StartNew (() => numbers [1]);     // Выход индекса за
                                                 // допустимые пределы
  childFactory.StartNew (() => { throw null; }); // Ссылка null
});
try { parent.Wait(); }
catch (AggregateException aex)
{
  aex.Flatten().Handle (ex => // Обратите внимание, что по-прежнему
                              // нужно вызывать Flatten
  {
    if (ex is DivideByZeroException)
    {
      Console.WriteLine ("Divide by zero");  // Деление на ноль
      return true;                           // Это исключение "обработано"
    }
    if (ex is IndexOutOfRangeException)
    {
      Console.WriteLine ("Index out of range");// Выход индекса
                                               // за допустимые пределы
      return true;                             // Это исключение "обработано"
    }
    return false;  // Все остальные исключения будут сгенерированы повторно
  });
}
```

# Параллельные коллекции

В версии .NET Framework 4.0 появился набор новых коллекций, определенных в пространстве имен `System.Collections.Concurrent`. Все они полностью безопасны в отношении потоков:

| Параллельная коллекция | Непараллельный эквивалент |
|---|---|
| `ConcurrentStack<T>` | `Stack<T>` |
| `ConcurrentQueue<T>` | `Queue<T>` |
| `ConcurrentBag<T>` | (отсутствует) |
| `ConcurrentDictionary<TKey,TValue>` | `Dictionary<TKey,TValue>` |

Параллельные коллекции оптимизированы для сценариев с высокой степенью параллелизма; тем не менее, они также могут быть полезны в ситуациях, когда требуется коллекция, безопасная к потокам (в качестве альтернативы применению блокировки к обычной коллекции). Однако с параллельными коллекциями связано несколько важных предостережений.

- По производительности традиционные коллекции превосходят параллельные коллекции во всех сценариях кроме тех, которые характеризуются высокой степенью параллелизма.
- Безопасная к потокам коллекция вовсе не гарантирует, что код, в котором она используется, будет безопасным в отношении потоков (см. разделы, посвященные безопасности к потокам, в главе 22).
- Если вы производите перечисление параллельной коллекции, в то время как другой поток ее модифицирует, то никаких исключений не возникает – взамен вы получите смесь старого и нового содержимого.
- Параллельной версии `List<T>` не существует.
- Параллельные классы стека, очереди и пакета внутренне реализованы с помощью связных списков. Это делает их менее эффективными в плане потребления памяти, чем непараллельные классы `Stack` и `Queue`, но лучшими для параллельного доступа, т.к. связные списки способствуют построению реализаций с низкой блокировкой или вообще без таковой. (Причина в том, что вставка узла в связный список требует обновления лишь пары ссылок, тогда как вставка элемента в структуру, подобную `List<T>`, может привести к перемещению тысяч существующих элементов.)

Другими словами, параллельные коллекции не являются простыми сокращениями для применения обычных коллекций с блокировками. В качестве демонстрации, если запустить следующий код в *одиночном* потоке:

```
var d = new ConcurrentDictionary<int,int>();
for (int i = 0; i < 1000000; i++) d[i] = 123;
```

то он выполнится в три раза медленнее, чем такой код:

```
var d = new Dictionary<int,int>();
for (int i = 0; i < 1000000; i++) lock (d) d[i] = 123;
```

(Тем не менее, *чтение* из `ConcurrentDictionary` будет быстрым, потому что операции чтения свободны от блокировок.)

Параллельные коллекции также отличаются от традиционных коллекций тем, что они открывают доступ к специальным методам, которые предназначены для выполнения атомарных операций типа "проверить и действовать", подобных TryPop. Большинство таких методов унифицировано посредством интерфейса IProducerConsumerCollection<T>.

## IProducerConsumerCollection<T>

Коллекция производителей/потребителей является одной из тех, для которых предусмотрены два главных сценария использования:

- добавление элемента (действие "производителя");
- извлечение элемента с его удалением (действие "потребителя").

Классическими примерами являются стеки и очереди. Коллекции производителей/потребителей играют важную роль в параллельном программировании, т.к. они способствуют построению эффективных реализаций, свободных от блокировок.

Интерфейс IProducerConsumerCollection<T> представляет безопасную к потокам коллекцию производителей/потребителей и реализован следующими классами:

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentBag<T>

Интерфейс IProducerConsumerCollection<T> расширяет ICollection, добавляя перечисленные ниже методы:

```
void CopyTo (T[] array, int index);
T[] ToArray();
bool TryAdd (T item);
bool TryTake (out T item);
```

Методы TryAdd и TryTake проверяют, может ли быть выполнена операция добавления/удаления, и если может, тогда производят добавление/удаление. Проверка и действие выполняются атомарно, устраняя необходимость в блокировке, к которой пришлось бы прибегнуть в случае традиционной коллекции:

```
int result;
lock (myStack) if (myStack.Count > 0) result = myStack.Pop();
```

Метод TryTake возвращает false, если коллекция пуста. Метод TryAdd всегда выполняется успешно и возвращает true в предоставленных трех реализациях. Однако если вы разрабатываете собственную параллельную коллекцию, в которой дубликаты запрещены, то обеспечите возврат методом TryAdd значения false, когда заданный элемент уже существует (примером может служить реализация параллельного *набора*).

Конкретный элемент, который TryTake удаляет, определяется подклассом:

- в случае стека TryTake удаляет элемент, добавленный позже всех других;
- в случае очереди TryTake удаляет элемент, добавленный раньше всех других;
- в случае пакета TryTake удаляет любой элемент, который может быть удален наиболее эффективно.

Три конкретных класса главным образом реализуют методы TryTake и TryAdd явно, делая доступной ту же самую функциональность через открытые методы с более специфичными именами, такими как TryDequeue и TryPop.

## ConcurrentBag<T>

Класс ConcurrentBag<T> хранит *неупорядоченную* коллекцию объектов (с разрешенными дубликатами). Класс ConcurrentBag<T> подходит в ситуациях, когда не имеет значения, какой элемент будет получен при вызове Take или TryTake.

Преимущество ConcurrentBag<T> перед параллельной очередью или стеком связано с тем, что метод Add пакета не допускает почти никаких состязаний, когда вызывается многими потоками одновременно. В отличие от него вызов Add параллельно на очереди или стеке приводит к *некоторым* состязаниям (хотя и намного меньшим, чем при блокировании *непараллельной* коллекции). Вызов Take на параллельном пакете также очень эффективен — до тех пор, пока каждый поток не извлекает большее количество элементов, чем он добавил с помощью Add.

Внутри параллельного пакета каждый поток получает свой закрытый связный список. Элементы добавляются в закрытый список, который принадлежит потоку, вызывающему Add, что устраняет состязания. Когда производится перечисление пакета, перечислитель проходит по закрытым спискам всех потоков, выдавая каждый из их элементов по очереди. Когда вызывается метод Take, пакет сначала просматривает закрытый список текущего потока. Если в нем имеется хотя бы один элемент[1], то задача может быть завершена легко и без состязаний. Но если этот список пуст, то пакет должен "позаимствовать" элемент из закрытого списка другого потока, что потенциально может привести к состязаниям.

Таким образом, чтобы соблюсти точность, вызов Take дает элемент, который был добавлен позже других в данном потоке; если же в этом потоке элементов нет, тогда Take дает последний добавленный элемент в другом потоке, выбранном произвольно.

Параллельные пакеты идеальны, когда параллельная операция на коллекции в основном состоит из добавления элементов посредством Add — или когда количество вызовов Add и Take сбалансировано в рамках потока. Пример первой ситуации приводился ранее во время применения метода Parallel.ForEach при реализации параллельной программы проверки орфографии:

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();
Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

Параллельный пакет может оказаться неудачным выбором для очереди производителей/потребителей, поскольку элементы добавляются и удаляются *разными* потоками.

# BlockingCollection<T>

В случае вызова метода TryTake на любой коллекции производителей/потребителей, рассмотренной в предыдущем разделе:

```
ConcurrentStack<T>,        ConcurrentQueue<T>,        ConcurrentBag<T>,
```

он возвращает false, если коллекция пуста. Иногда в таком сценарии полезнее организовать *ожидание*, пока элемент не станет доступным.

Вместо перегрузки методов TryTake для обеспечения такой функциональности (что привело бы к перенасыщению членами после предоставления возможности

[1] Из-за деталей реализации на самом деле должны существовать хотя бы два элемента, чтобы полностью избежать состязаний.

работы с признаками отмены и тайм-аутами) проектировщики PFX инкапсулировали ее в класс-оболочку по имени `BlockingCollection<T>`. Блокирующая коллекция может содержать внутри любую коллекцию, которая реализует интерфейс `IProducerConsumerCollection<T>`, и позволяет получать с помощью метода `Take` элемент из внутренней коллекции, обеспечивая блокирование, когда доступных элементов нет.

Блокирующая коллекция также позволяет ограничивать общий размер коллекции, блокируя *производителя*, если этот размер превышен. Коллекция, ограниченная в подобной манере, называется *ограниченной блокирующей коллекцией*.

Для использования класса `BlockingCollection<T>` необходимо выполнить описанные ниже шаги.

1. Создать экземпляр класса, дополнительно указывая помещаемую внутрь реализацию `IProducerConsumerCollection<T>` и максимальный размер (границу) коллекции.
2. Вызывать метод `Add` или `TryAdd` для добавления элементов во внутреннюю коллекцию.
3. Вызывать метод `Take` или `TryTake` для удаления (потребления) элементов из внутренней коллекции.

Если конструктор вызван без передачи ему коллекции, то автоматически будет создан экземпляр `ConcurrentQueue<T>`. Методы производителя и потребителя позволяют указывать признаки отмены и тайм-ауты. Методы `Add` и `TryAdd` могут блокироваться, если размер коллекции ограничен; методы `Take` и `TryTake` блокируются на время, пока коллекция пуста.

Еще один способ потребления элементов предполагает вызов метода `GetConsumingEnumerable`. Он возвращает (потенциально) бесконечную последовательность, которая выдает элементы по мере того, как они становятся доступными. Чтобы принудительно завершить такую последовательность, необходимо вызвать `CompleteAdding`: этот метод также предотвращает помещение в очередь дальнейших элементов.

Кроме того, класс `BlockingCollection` предоставляет статические методы под названиями `AddToAny` и `TakeFromAny`, которые позволяют добавлять и получать элемент, указывая несколько блокирующих коллекций. Действие затем будет выполнено первой коллекцией, которая способна обслужить данный запрос.

## Реализация очереди производителей/потребителей

Очередь производителей/потребителей – структура, полезная как при параллельном программировании, так и в общих сценариях параллелизма. Ниже описаны основные аспекты ее работы.

- Очередь настраивается для описания элементов работы или данных, над которыми выполняется работа.
- Когда задача должна выполниться, она ставится в очередь, а вызывающий код занимается другой работой.
- Один или большее число рабочих потоков функционируют в фоновом режиме, извлекая и запуская элементы из очереди.

Очередь производителей/потребителей обеспечивает точный контроль над тем, сколько рабочих потоков выполняется за раз, что полезно для ограничения эксплуатации не только ЦП, но также и других ресурсов. Скажем, если задачи выполняют интенсивные

операции дискового ввода-вывода, то можно ограничить параллелизм, не истощая операционную систему и другие приложения. На протяжении времени жизни очереди можно также динамически добавлять и удалять рабочие потоки. Пул потоков CLR сам представляет собой разновидность очереди производителей/потребителей, которая оптимизирована для кратко выполняющихся заданий с интенсивными вычислениями.

Очередь производителей/потребителей обычно хранит элементы данных, на которых выполняется (одна и та же) задача. Например, элементами данных могут быть имена файлов, а задача может осуществлять шифрование содержимого таких файлов. С другой стороны, применяя делегаты в качестве элементов, можно построить более универсальную очередь производителей/потребителей, где каждый элемент способен делать все что угодно.

В статье "Parallel Programming" ("Параллельное программирование") по адресу `http://albahari.com/threading` мы показываем, как реализовать очередь производителей/потребителей с нуля, используя событие `AutoResetEvent` (а также впоследствии методы `Wait` и `Pulse` класса `Monitor`). Тем не менее, начиная с версии .NET Framework 4.0, написание очереди производителей/потребителей с нуля стало необязательным, т.к. *б*ольшая часть функциональности предлагается классом `BlockingCollection<T>`. Вот как его задействовать:

```
public class PCQueue : IDisposable
{
  BlockingCollection<Action> _taskQ = new BlockingCollection<Action>();
  public PCQueue (int workerCount)
  {
    // Создать и запустить отдельный объект Task для каждого потребителя:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }
  public void Enqueue (Action action) { _taskQ.Add (action); }
  void Consume()
  {
    // Эта перечисляемая последовательность будет блокироваться,
    // когда нет доступных элементов, и заканчиваться, когда вызван
    // метод CompleteAdding.
    foreach (Action action in _taskQ.GetConsumingEnumerable())
      action(); // Выполнить задачу.
  }
  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

Поскольку конструктору `BlockingCollection` ничего не передается, он автоматически создает параллельную очередь. Если бы ему был передан объект `ConcurrentStack`, тогда мы получили бы в итоге стек производителей/потребителей.

### Использование задач

Только что написанная очередь производителей/потребителей не является гибкой, т.к. мы не можем отслеживать элементы работы после их помещения в очередь. Очень полезными были бы следующие возможности:

- знать, когда элемент работы завершается (и ожидать его посредством `await`);
- отменять элемент работы;
- элегантно обрабатывать любые исключения, которые сгенерированы тем или иным элементом работы.

Идеальное решение предусматривало бы возможность возвращения методом Enqueue какого-то объекта, снабжающего нас описанной выше функциональностью. К счастью, уже существует класс, делающий в точности то, что нам нужно – это Task, объект которого можно либо сгенерировать с помощью TaskCompletionSource, либо создать напрямую (получив незапущенную или *холодную* задачу):

```
public class PCQueue : IDisposable
{
  BlockingCollection<Task> _taskQ = new BlockingCollection<Task>();
  public PCQueue (int workerCount)
  {
    // Создать и запустить отдельный объект Task для каждого потребителя:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }
  public Task Enqueue (Action action, CancellationToken cancelToken
                                      = default (CancellationToken))
  {
    var task = new Task (action, cancelToken);
    _taskQ.Add (task);
    return task;
  }
  public Task<TResult> Enqueue<TResult> (Func<TResult> func,
              CancellationToken cancelToken = default (CancellationToken))
  {
    var task = new Task<TResult> (func, cancelToken);
    _taskQ.Add (task);
    return task;
  }
  void Consume()
  {
    foreach (var task in _taskQ.GetConsumingEnumerable())
      try
      {
         if (!task.IsCanceled) task.RunSynchronously();
      }
      catch (InvalidOperationException) { }  // Условие состязаний
  }
  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

В методе Enqueue мы помещаем в очередь и возвращаем вызывающему коду задачу, которая создана, но не запущена.

В методе Consume мы запускаем эту задачу синхронно в потоке потребителя. Мы перехватываем исключение InvalidOperationException, чтобы обработать маловероятную ситуацию, когда задача будет отменена в промежутке между проверкой, не отменена ли она, и ее запуском.

Ниже показано, как можно применять класс PCQueue:

```
var pcQ = new PCQueue (2);  // Максимальная степень параллелизма равна 2
string result = await pcQ.Enqueue (() => "That was easy!");
...
```

Следовательно, мы имеем все преимущества задач – распространение исключений, возвращаемые значения и возможность отмены – и в то же время обладаем полным контролем над их планированием.

# 24

# Домены приложений

*Домен приложения* — это единица изоляции времени выполнения, внутри которой запускается программа .NET. Он предоставляет границы управляемой памяти, контейнер для загруженных сборок и параметров конфигурации приложения, а также контуры коммуникационных границ для распределенных приложений.

Каждый процесс .NET обычно размещает только один домен приложения — стандартный домен, автоматически создаваемый средой CLR при запуске процесса. В рамках того же самого процесса также возможно (и временами полезно) создавать дополнительные домены приложений, что обеспечивает изоляцию и помогает избежать накладных расходов и усложнения коммуникаций, связанных с наличием отдельного процесса. Такой подход удобен в сценариях тестирования нагрузки и исправления приложений, а также для реализации надежных механизмов восстановления после ошибок.

Материалы в настоящей главе не имеют никакого отношения к приложениям UWP и .NET Core, в которых разрешен доступ только к одному домену приложения.

## Архитектура доменов приложений

На рис. 24.1 представлены архитектуры для приложения с единственным доменом, приложения с несколькими доменами и типичного распределенного клиент-серверного приложения. В большинстве случаев процессы, заключающие домены приложений, создаются неявно операционной системой — когда пользователь дважды щелкает на исполняемом файле .NET или запускает Windows-службу. Тем не менее, домен приложения может также размещаться в других процессах, таких как IIS или SQL Server через интеграцию с CLR.

В случае простого исполняемого файла процесс заканчивается, когда стандартный домен приложения завершает выполнение. Однако на таких хостах, как IIS или SQL Server, временем жизни доменов управляет процесс, создавая и уничтожая домены приложений .NET, как он считает нужным.

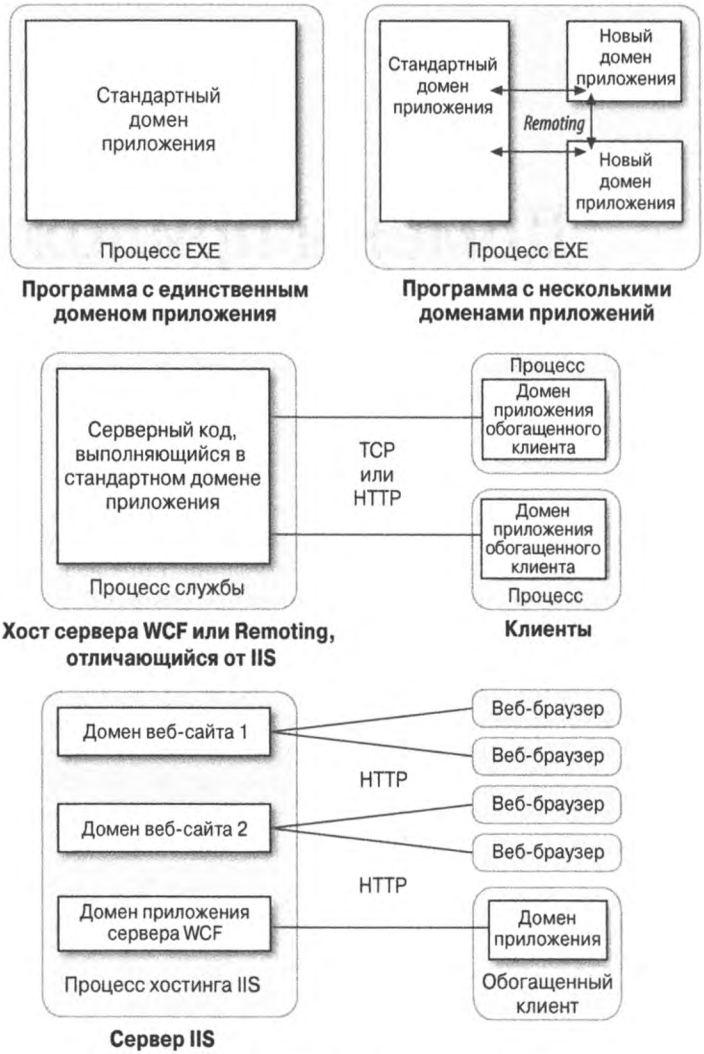



***Рис. 24.1.** Архитектуры доменов приложений*

# Создание и уничтожение доменов приложений

Дополнительные домены приложений в процессе создаются и уничтожаются с помощью статических методов `AppDomain.CreateDomain` и `AppDomain.Unload`. В следующем примере сборка `test.exe` выполняется в изолированном домене приложения, который затем выгружается:

```
static void Main()
{
  AppDomain newDomain = AppDomain.CreateDomain ("New Domain");
  newDomain.ExecuteAssembly ("test.exe");
  AppDomain.Unload (newDomain);
}
```

Обратите внимание, что когда стандартный домен приложения (созданный средой CLR во время начальной запуска) выгружается, все другие домены приложений выгружаются автоматически, и приложение закрывается. Для выяснения, является ли домен стандартным, используется метод `IsDefaultAppDomain` класса `AppDomain`.

Класс `AppDomainSetup` предоставляет параметры, которые могут быть заданы для нового домена. Ниже перечислены наиболее полезные свойства этого класса:

```
public string ApplicationName { get; set; }      // "Дружественное" имя
public string ApplicationBase { get; set; }      // Базовая папка

public string ConfigurationFile { get; set; }
public string LicenseFile       { get; set; }

// Для оказания помощи с автоматическим распознаванием сборок:
public string PrivateBinPath      { get; set; }
public string PrivateBinPathProbe { get; set; }
```

Свойство `ApplicationBase` управляет базовым каталогом домена приложения, применяемым в качестве корня при автоматическом поиске сборок. В стандартном домене приложения это папка главной исполняемой сборки. В новом домене, который создается, базовый каталог может находиться где угодно:

```
AppDomainSetup setup = new AppDomainSetup();
setup.ApplicationBase = @"c:\MyBaseFolder";
AppDomain newDomain = AppDomain.CreateDomain ("New Domain", null, setup);
```

Новый домен также можно подписать на получение событий распознавания сборок, определенных в домене инициатора:

```
static void Main()
{
  AppDomain newDomain = AppDomain.CreateDomain ("test");
  newDomain.AssemblyResolve += new ResolveEventHandler (FindAssem);
  ...
}

static Assembly FindAssem (object sender, ResolveEventArgs args)
{
  ...
}
```

Прием допустим при условии, что обработчик событий является статическим методом, который определен в типе, доступном обоим доменам. Тогда среда CLR имеет возможность запускать такой обработчик событий в подходящем домене. В рассматриваемом примере `FindAssem` будет выполняться из `newDomain`, хотя подписание осуществлялось из стандартного домена.

Свойство `PrivateBinPath` представляет собой список подкаталогов ниже базового каталога с разделителем в форме точки с запятой, где среда CLR должна искать сборки. (Как и базовая папка приложения, данное свойство может устанавливаться только перед запуском домена приложения.) Возьмем структуру каталогов, в которой программа имеет единственный исполняемый файл (и возможно конфигурационный файл) в своей базовой папке и все ссылочные сборки в подпапках:

```
c:\MyBaseFolder\              -- Запускаемый исполняемый файл
            \bin
            \bin\v1.23        -- Последние DLL-файлы сборок
            \bin\plugins      -- Дополнительные DLL-файлы
```

Вот как должен быть настроен домен приложения для использования показанной структуры каталогов:

```
AppDomainSetup setup  = new AppDomainSetup();
setup.ApplicationBase = @"c:\MyBaseFolder";
setup.PrivateBinPath  = @"bin\v1.23;bin\plugins";
AppDomain d = AppDomain.CreateDomain ("New Domain", null, setup);
d.ExecuteAssembly (@"c:\MyBaseFolder\Startup.exe");
```

Обратите внимание, что свойство `PrivateBinPath` всегда является относительным и находится ниже базовой папки приложения. Указывать абсолютные пути не разрешено. Класс `AppDomain` также предлагает свойство `PrivateBinPathProbe`, которое в случае установки во что-либо, отличающееся от пустой строки, исключает сам базовый каталог из пути поиска сборок. (Причина выбора для свойства `PrivateBinPathProbe` типа `string`, а не `bool`, обусловлена поддержкой совместимости с COM.)

Прямо перед выгрузкой домена приложения, отличающегося от стандартного, инициируется событие `DomainUnload`. Его можно применять для выполнения логики очистки: выгрузка домена (и при необходимости приложения в целом) откладывается вплоть до завершения всех обработчиков событий `DomainUnload`.

Непосредственно перед закрытием самого приложения инициируется событие `ProcessExit` на всех загруженных доменах приложений (включая стандартный домен). В отличие от события `DomainUnload` выполнение обработчиков событий `ProcessExit` нормируется по времени: стандартный хост CLR дает обработчикам событий по две секунды на домен и три секунды совокупно перед завершением их потоков.

# Использование нескольких доменов приложений

Несколько доменов приложений применяются в следующих основных случаях:

- обеспечение изоляции, подобной изоляции процессов, с минимальными накладными расходами;
- предоставление файлам сборок возможности быть выгруженными без перезапуска процесса.

Когда дополнительные домены приложений созданы внутри того же самого процесса, среда CLR обеспечивает для каждого из них уровень изоляции, сходный с таковым при выполнении в отдельных процессах. Это означает, что каждый домен имеет отдельную память, и объекты в одном домене не могут мешать объектам из другого домена. Кроме того, статические члены того же самого класса имеют независимые значения в каждом домене. Инфраструктура ASP.NET использует точно такой же подход при предоставлении множеству сайтов возможности выполняться в разделяемом процессе, не влияя друг на друга.

В ASP.NET домены приложений создаются инфраструктурой — безо всякого вмешательства с вашей стороны. Однако бывают ситуации, когда можно извлечь преимущества из явного создания множества доменов внутри одиночного процесса. Предположим, что вы написали собственную систему аутентификации и в качестве части модульного тестирования хотите провести нагрузочное тестирование серверного кода, эмулируя одновременный вход в систему 20 клиентов. Для эмуляции 20 параллельных входов доступны три варианта:

- запустить 20 отдельных процессов, 20 раз вызвав метод `Process.Start`;
- запустить 20 потоков в том же самом процессе и домене;
- запустить 20 потоков в том же самом процессе, но каждый в собственном домене приложения.

Первый вариант является утомительным и ресурсоемким. Кроме того, взаимодействовать с каждым отдельным процессом будет нелегко, если ему необходимо предоставлять более специфичные инструкции о том, что он должен делать.

Второй вариант полагается на то, что код клиентской стороны безопасен в отношении потоков, а это маловероятно — особенно, если для хранения текущего состояния аутентификации применяются статические переменные. И добавление блокировки вокруг кода клиентской стороны будет препятствовать параллельному выполнению, которое необходимо для проведения нагрузочного тестирования сервера.

Третий вариант идеален. В данном случае каждый поток сохраняется изолированным — с независимым состоянием — и все же остается легко достижимым для размещающей программы.

Еще одна причина создавать отдельный домен приложения — позволить сборкам выгружаться, не завершая процесс. Это вытекает из того факта, что выгрузить сборку можно только путем закрытия домена приложения, который ее загрузил. Ситуация становится проблематичной, если сборка была загружена в стандартный домен, т.к. его закрытие означает закрытие самого приложения. Файл сборки блокируется, пока сборка загружена, и потому его нельзя исправить или заменить. Загрузка сборок в отдельный домен приложения, который может быть уничтожен, позволяет обойти данную проблему, а также помогает сократить отпечаток в памяти приложения, иногда нуждающегося в загрузке больших сборок.

### Атрибут `LoaderOptimization`

По умолчанию сборки, которые загружаются в явно созданный домен приложения, повторно обрабатываются JIT-компилятором. В их число входят:

- сборки, которые уже были обработаны JIT-компилятором в домене вызывающего компонента;
- сборки, для которых был сгенерирован машинный образ с помощью инструмента `ngen.exe`;
- все сборки .NET Framework (кроме `mscorlib`).

В результате производительности может быть нанесен серьезный урон, особенно в случае частого создания и выгрузки доменов приложений, которые ссылаются на крупные сборки .NET Framework. Обходной прием предусматривает присоединение к методу главной точки входа в программу следующего атрибута:

```
[LoaderOptimization (LoaderOptimization.MultiDomainHost)]
```

Данный атрибут сообщает CLR о необходимости загрузки сборок из GAC как *нейтральных к домену*, поэтому на обработку принимаются машинные образы, а образы JIT разделяются между доменами приложений. Обычно такое решение идеально, потому что GAC включает все сборки .NET Framework (и возможно некоторые неизменяемые части вашего приложения).

Можно пойти еще дальше и указать в атрибуте значение `LoaderOptimization.MultiDomain`, что сообщит *всем* сборкам о необходимости загружаться нейтрально к домену (исключая сборки, которые загружаются за пределами нормального механизма распознавания сборок). Однако такой подход нежелателен, если нужно, чтобы сборки выгружались их доменами. Нейтральная к домену сборка разделяется всеми доменами и потому не может быть выгружена, пока не закончится родительский процесс.

## Использование DoCallBack

Давайте вернемся к самому базовому сценарию с несколькими доменами:

```
static void Main()
{
  AppDomain newDomain = AppDomain.CreateDomain ("New Domain");
  newDomain.ExecuteAssembly ("test.exe");
  AppDomain.Unload (newDomain);
}
```

Вызов метода ExecuteAssembly на отдельном домене удобен, но предлагает мало возможностей взаимодействия с доменом. Он также требует, чтобы целевая сборка являлась исполняемой, и фиксирует вызывающий компонент на одной точке входа. Единственный способ обеспечения гибкости – прибегнуть к такому подходу, как передача строки аргументов исполняемой сборке.

Более мощный подход предполагает применение метода DoCallBack класса AppDomain. Он выполняет в другом домене приложения метод заданного типа, чья сборка автоматически загружается в домен (среда CLR будет знать, где находится сборка, если текущий домен может сослаться на нее). В приведенном ниже примере метод текущего выполняемого класса запускается в новом домене:

```
class Program
{
  static void Main()
  {
    AppDomain newDomain = AppDomain.CreateDomain ("New Domain");
    newDomain.DoCallBack (new CrossAppDomainDelegate (SayHello));
    AppDomain.Unload (newDomain);
  }
  static void SayHello()
  {
    Console.WriteLine ("Hi from " + AppDomain.CurrentDomain.FriendlyName);
  }
}
```

Пример работает из-за того, что делегат ссылается на статический метод, т.е. указывает на тип, а не на экземпляр. Это делает делегат "независимым от домена" или гибким. Он может запускаться в любом домене одним и тем же способом, т.к. ничто его не привязывает к исходному домену. Метод DoCallBack допускается также использовать с делегатом, ссылающимся на метод экземпляра. Тем не менее, среда CLR попытается применить семантику Remoting (рассматривается далее в главе), которая в данном случае оказывается противоположной тому, что нам нужно.

# Мониторинг доменов приложений

Начиная с версии .NET Framework 4.0, можно проводить мониторинг потребления памяти и центрального процессора (ЦП) определенным доменом приложения. Чтобы это работало, сначала потребуется включить мониторинг домена приложения:

```
AppDomain.MonitoringIsEnabled = true;
```

Данный код включает мониторинг текущего домена. После включения отключить мониторинг не получится – установка свойства MonitoringIsEnabled в false приводит к генерации исключения.

Другой способ включения мониторинга домена предусматривает использование конфигурационного файла приложения. Добавьте в него следующий элемент:

```
<configuration>
  <runtime>
    <appDomainResourceMonitoring enabled="true"/>
  </runtime>
</configuration>
```

В результате включится мониторинг для всех доменов приложений.

Затем можно запрашивать сведения о потреблении ЦП и памяти доменом приложения через перечисленные ниже три свойства экземпляра `AppDomain`:

- `MonitoringTotalProcessorTime`
- `MonitoringTotalAllocatedMemorySize`
- `MonitoringSurvivedMemorySize`

Первые два свойства возвращают показатели *общего* потребления ЦП и управляемой памяти, выделенной доменом с момента его запуска. (Такие цифры могут только возрастать, но никогда не уменьшаться.) Третье свойство возвращает действительное потребление управляемой памяти доменом на момент последней сборки мусора.

Обращаться к упомянутым свойствам можно из того же самого либо из другого домена.

# Домены и потоки

Когда вызывается метод в другом домене приложения, выполнение блокируется до тех пор, пока данный метод не завершит свою работу – точно так же, как если бы метод вызывался в текущем домене. Хотя обычно такое поведение является желательным, существуют случаи, когда метод нужно запустить параллельно. Это можно делать с помощью многопоточности.

Ранее говорилось о применении нескольких доменов приложений для эмуляции параллельного входа в систему 20 клиентов в рамках тестирования системы аутентификации. Обеспечив вход каждого клиента в отдельном домене приложения, каждый клиент будет изолирован и не сможет влиять на других клиентов через статические члены класса. Для реализации данного примера мы должны вызвать метод `Login` в 20 параллельных потоках, каждый из которых находится в собственном домене приложения:

```
class Program
{
  static void Main()
  {
    // Создать 20 доменов и 20 потоков.
    AppDomain[] domains = new AppDomain [20];
    Thread[] threads = new Thread [20];

    for (int i = 0; i < 20; i++)
    {
      domains [i] = AppDomain.CreateDomain ("Client Login " + i);
      threads [i] = new Thread (LoginOtherDomain);
    }
```

```
    // Запустить все потоки, передавая каждому его домен приложения.
    for (int i = 0; i < 20; i++) threads [i].Start (domains [i]);

    // Ожидать завершения потоков.
    for (int i = 0; i < 20; i++) threads [i].Join();

    // Выгрузить домены приложений.
    for (int i = 0; i < 20; i++) AppDomain.Unload (domains [i]);
    Console.ReadLine();
  }

  // Параметризированный запуск потока с передачей домена,
  // в котором он должен выполняться.
  static void LoginOtherDomain (object domain)
  {
    ((AppDomain) domain).DoCallBack (Login);
  }

  static void Login()
  {
    Client.Login ("Joe", "");
    Console.WriteLine ("Logged in as: " + Client.CurrentUser + " on " +
      AppDomain.CurrentDomain.FriendlyName);
  }
}

class Client
{
  // Статическое поле, через которое входы клиентов могут влиять друг
  // на друга, если выполняются в одном и том же домене приложения.
  public static string CurrentUser = "";

  public static void Login (string name, string password)
  {
   if (CurrentUser.Length == 0)  // Если вход еще не совершен...
    {
      // Пауза для эмулирования аутентификации...
      Thread.Sleep (500);
      CurrentUser = name;  // Зафиксировать факт прохождения аутентификации.
    }
  }
}

// Вывод:
Logged in as: Joe on Client Login 0
Logged in as: Joe on Client Login 1
Logged in as: Joe on Client Login 4
Logged in as: Joe on Client Login 2
Logged in as: Joe on Client Login 3
Logged in as: Joe on Client Login 5
Logged in as: Joe on Client Login 6
...
```

Более подробные сведения о многопоточности ищите в главе 22.

# Разделение данных между доменами

## Разделение данных через ячейки

Домены приложений могут использовать именованные ячейки для разделения данных, как демонстрируется в следующем примере:

```
class Program
{
  static void Main()
  {
    AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

    // Записать в ячейку по имени Message - подойдет любой строковый ключ.
    newDomain.SetData ("Message", "guess what...");

    newDomain.DoCallBack (SayMessage);
    AppDomain.Unload (newDomain);
  }

  static void SayMessage()
  {
    // Читать из ячейки данных Message.
    Console.WriteLine (AppDomain.CurrentDomain.GetData ("Message"));
  }
}

// Вывод:
guess what...
```

Ячейка создается автоматически при первом обращении к ней. Данные, участвующие в коммуникациях (строка "guess what ..." в рассмотренном примере), должны либо быть *сериализируемыми* (см. главу 17), либо основанными на `MarshalByRefObject`. Если данные поддерживают сериализацию (наподобие строки в настоящем примере), то они копируются в другой домен приложения. Если они являются производными от класса `MarshalByRefObject`, тогда применяется семантика Remoting.

## Использование Remoting внутри процесса

Наиболее гибкий способ взаимодействия с другим доменом приложения предполагает создание объектов *в другом домене* через прокси. Это называется *технологией Remoting*.

Класс, участвующий в Remoting, должен быть унаследован от класса `MarshalByRefObject`. Тогда для создания удаленного объекта клиент вызывает метод `CreateInstance`*XXX* на удаленном домене приложения.

Следующий код создает объект типа `Foo` в другом домене приложения, после чего вызывает его метод `SayHello`:

```
class Program
{
  static void Main()
  {
    AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

    Foo foo = (Foo) newDomain.CreateInstanceAndUnwrap (
                    typeof (Foo).Assembly.FullName,
                    typeof (Foo).FullName);

    Console.WriteLine (foo.SayHello());
```

```
    AppDomain.Unload (newDomain);
    Console.ReadLine();
  }
}

public class Foo : MarshalByRefObject
{
  public string SayHello()
    => "Hello from " + AppDomain.CurrentDomain.FriendlyName;

  // Это обеспечивает существование объекта столько, сколько желает клиент.
  public override object InitializeLifetimeService() => null;
}
```

Когда объект `foo` создается в другом домене приложения (который называется "удаленным" доменом), мы не можем получить обратно прямую ссылку на этот объект, поскольку домены приложений изолированы друг от друга. Взамен мы получаем прозрачный прокси; прозрачный он потому, что *выглядит* так, как если бы он был прямой ссылкой на удаленный объект. При последующем вызове метода `SayHello` объекта `foo` "за кулисами" конструируется сообщение, которое направляется "удаленному" домену приложения, где затем выполняется с реальным объектом `foo`. Ситуация похожа на звонок по телефону: вы разговариваете не с реальным человеком, а с куском пластика, который действует в качестве прозрачного прокси для этого человека. Любое возвращаемое значение переводится в сообщение и отправляется обратно вызывающему коду.

До появления в версии .NET Framework 3.0 инфраструктуры Windows Communication Foundation (WCF) технология Remoting была одной из двух главных технологий для написания распределенных приложений (другой выступала технология веб-служб (Web Services)). В распределенном приложении Remoting на каждой стороне явно настраивался коммуникационный канал HTTP или TCP/IP, который позволял взаимодействию пересекать границы процесса и сети.

Хотя WCF превосходит Remoting при построении распределенных приложений, за Remoting по-прежнему осталась ниша взаимодействия между доменами внутри процесса. Преимущество применения технологии Remoting в данном сценарии связано с тем, что она не требует конфигурирования, т.к. коммуникационный канал создается автоматически (быстрый канал в памяти) и не подразумевает какой-либо регистрации типов. Вы просто начинаете пользоваться Remoting и все.

Методы объекта `Foo` могут возвращать дополнительные экземпляры `MarshalByRefObject`, и в таком случае при вызове этих методов генерируются дополнительные прозрачные прокси. Методы объекта `Foo` могут также принимать экземпляры `MarshalByRefObject` в качестве аргументов — в данной ситуации технология Remoting работает в обратном направлении. Вызывающий код будет удерживать "удаленный" объект, а вызываемый будет иметь прокси.

Как и маршализация объектов по ссылке, домены приложений могут обмениваться скалярными значениями или любым *сериализируемым* объектом. Тип является сериализируемым, если он имеет атрибут `Serializable` либо реализует интерфейс `ISerializable`. Тогда при пересечении границы домена приложения возвращается полная копия такого объекта, а не прокси. Другими словами, объект маршализируется по *значению*, а не по ссылке.

Функционирование Remoting внутри одного и того же процесса активизируется клиентом, означая, что CLR не пытается разделять или повторно использовать удаленно созданные объекты в том же самом или в другом клиенте. Говоря иначе, если клиент создает два объекта Foo, то два объекта будут созданы в удаленном домене и два прокси – в клиентском домене. Это обеспечивает наиболее естественную семантику объектов, однако означает, что удаленный домен зависит от сборщика мусора клиента: объект foo в удаленном домене освобождается из памяти, только когда сборщик мусора клиента решит, что foo (прокси) больше не применяется. Если в клиентском домене происходит аварийный отказ, то объект foo может никогда не быть освобожденным. Чтобы защититься от такого сценария, среда CLR предоставляет основанный на аренде механизм для управления временем жизни удаленно созданных объектов. Стандартное поведение удаленно созданных объектов предусматривает их самоуничтожение после того, как они не используются на протяжении пяти минут.

Поскольку в рассмотренном примере клиент выполняется в стандартном домене приложения, клиент не может себе позволить такую роскошь, как аварийный отказ. Если он закончит работу, то закончится целый процесс! Следовательно, пятиминутную аренду имеет смысл отключить. С этой целью и был переопределен метод InitializeLifetimeService – за счет возвращения аренды null удаленно созданные объекты уничтожаются, только когда обрабатываются сборщиком мусора на стороне клиента.

## Изолирование типов и сборок

В предшествующем примере мы удаленно создавали объект типа Foo следующим образом:

```
Foo foo = (Foo) newDomain.CreateInstanceAndUnwrap (
                typeof (Foo).Assembly.FullName,
                typeof (Foo).FullName);
```

Вот как выглядит сигнатура метода CreateInstanceAndUnwrap:

```
public object CreateInstanceAndUnwrap (string assemblyName,
                                       string typeName)
```

Из-за того, что данный метод принимает *имена* сборки и типа, а не объект Type, объект можно создать удаленно, не загружая его тип локально. Поступать так удобно, когда нужно избежать загрузки сборки типа в домен приложения, где находится вызывающий компонент.

Класс AppDomain также предлагает метод по имени CreateInstanceFromAndUnwrap. Ниже перечислены его отличия от метода CreateInstanceAndUnwrap:

- метод CreateInstanceAndUnwrap принимает *полностью заданное имя сборки* (см. главу 18);
- метод CreateInstanceFromAndUnwrap принимает *путь* либо *имя файла*.

В целях иллюстрации предположим, что был разработан текстовый редактор, который позволяет пользователю загружать и выгружать подключаемые модули, написанные третьими сторонами.

Первый шаг заключается в написании общей библиотеки, на которую будут ссылаться хост и подключаемые модули. В библиотеке будет определен интерфейс, описывающий то, что могут делать подключаемые модули. Вот простой пример:

```
namespace Plugin.Common
{
  public interface ITextPlugin
  {
    string TransformText (string input);
  }
}
```

Далее понадобится построить простой подключаемый модуль. Будем считать, что следующий код компилируется в сборку `AllCapitals.dll`:

```
namespace Plugin.Extensions
{
  public class AllCapitals : MarshalByRefObject, Plugin.Common.ITextPlugin
  {
    public string TransformText (string input) => input.ToUpper();
  }
}
```

Ниже показано, как реализовать хост, который загружает сборку `AllCapitals.dll` в отдельный домен приложения, вызывает метод `TransformText` с применением Remoting и затем выгружает этот домен приложения:

```
using System;
using System.Reflection;
using Plugin.Common;

class Program
{
  static void Main()
  {
    AppDomain domain = AppDomain.CreateDomain ("Plugin Domain");

    ITextPlugin plugin = (ITextPlugin) domain.CreateInstanceFromAndUnwrap
      ("AllCapitals.dll", "Plugin.Extensions.AllCapitals");

    // Вызвать метод TransformText, используя Remoting:
    Console.WriteLine (plugin.TransformText ("hello")); // "HELLO"

    AppDomain.Unload (domain);

    // Файл AllCapitals.dll теперь полностью выгружен
    // и может быть перемещен или удален.
  }
}
```

Поскольку данная программа взаимодействует с подключаемым модулем только через общий интерфейс `ITextPlugin`, типы в сборке `AllCapitals` никогда не загружаются в домен приложения, из которого производится вызов. Это поддерживает целостность вызывающего домена и гарантирует, что никакие блокировки не будут удерживаться на файлах сборок подключаемых модулей после выгрузки их домена.

## Обнаружение типов

В предыдущем примере реальному приложению потребовались бы какие-то средства обнаружения имен типов подключаемых модулей, таких как `Plugin.Extensions.AllCapitals`.

Достичь указанной цели можно, написав в *общей* сборке класс обнаружения, который использует рефлексию:

```
public class Discoverer : MarshalByRefObject
{
  public string[] GetPluginTypeNames (string assemblyPath)
  {
    List<string> typeNames = new List<string>();
    Assembly a = Assembly.LoadFrom (assemblyPath);
    foreach (Type t in a.GetTypes())
      if (t.IsPublic
        && t.IsMarshalByRef
        && typeof (ITextPlugin).IsAssignableFrom (t))
    {
      typeNames.Add (t.FullName);
    }
    return typeNames.ToArray();
  }
}
```

Загвоздка в том, что метод `Assembly.LoadFrom` загружает сборку в текущий домен приложения. Следовательно, данный метод должен вызываться *в домене подключаемого модуля*:

```
class Program
{
  static void Main()
  {
    AppDomain domain = AppDomain.CreateDomain ("Plugin Domain");

    Discoverer d = (Discoverer) domain.CreateInstanceAndUnwrap (
      typeof (Discoverer).Assembly.FullName,
      typeof (Discoverer).FullName);

    string[] plugInTypeNames = d.GetPluginTypeNames ("AllCapitals.dll");

    foreach (string s in plugInTypeNames)
      Console.WriteLine (s);  // Plugin.Extensions.AllCapitals

    ...
```

В сборке `System.AddIn.Contract` имеется API-интерфейс, который развивает продемонстрированные концепции в завершенную инфраструктуру для расширяемости программ. Он решает такие проблемы, как изоляция, поддержка версий, обнаружение, активация и т.д. Для получения дополнительной информации поищите блог "CLR Add-In Team Blog" на веб-сайте `http://blogs.msdn.com`.

# 25

# Способность к взаимодействию

В настоящей главе рассматриваются способы интеграции с низкоуровневыми (неуправляемыми) DLL-библиотеками и компонентами COM. Если не указано иначе, то упоминаемые в главе типы находятся либо в пространстве имен `System`, либо в пространстве имен `System.Runtime.InteropServices`.

## Обращение к низкоуровневым DLL-библиотекам

Технология *P/Invoke* (сокращение для Platform Invocation Services – службы вызова функций платформы) позволяет получать доступ к функциям, структурам и обратным вызовам в неуправляемых DLL-библиотеках. Например, рассмотрим функцию `MessageBox`, которая определена в DLL-библиотеке Windows по имени `user32.dll` следующим образом:

```
int MessageBox (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Эту функцию можно вызывать напрямую, объявив статический метод с тем же именем, применив ключевое слово `extern` и добавив атрибут `DllImport`:

```
using System;
using System.Runtime.InteropServices;

class MsgBoxTest
{
  [DllImport("user32.dll")]
  static extern int MessageBox (IntPtr hWnd, string text, string caption,
                                int type);
  public static void Main()
  {
    MessageBox (IntPtr.Zero,
                "Please do not press this again.", "Attention", 0);
  }
}
```

Классы `MessageBox` в пространствах имен `System.Windows` и `System.Windows.Forms` сами вызывают подобные неуправляемые методы.

Среда CLR включает маршализатор, которому известно, как преобразовывать параметры и возвращаемые значения между типами .NET и неуправляемыми типами. В приведенном примере параметры `int` транслируются прямо в четырехбайтовые целые числа, которые ожидает функция, а строковые параметры преобразуются в массивы двухбайтовых символов Unicode, завершающиеся символом `null`. Структура `IntPtr` предназначена для инкапсуляции неуправляемого дескриптора и занимает 32 бита на 32-разрядных платформах и 64 бита на 64-разрядных платформах.

# Маршализация типов

## Маршализация общих типов

На неуправляемой стороне для представления необходимого типа данных может существовать более одного способа. Скажем, строка может содержать однобайтовые символы ANSI или двухбайтовые символы Unicode и предваряться значением длины в качестве префикса, завершаться символом `null` либо иметь фиксированную длину. С помощью атрибута `MarshalAs` маршализатору CLR сообщается используемый вариант, так что он обеспечит корректную трансляцию. Ниже показан пример:

```
[DllImport("...")]
static extern int Foo ( [MarshalAs (UnmanagedType.LPStr)] string s );
```

Перечисление `UnmanagedType` включает все типы Win32 и COM, распознаваемые маршализатором. В этом случае маршализатору указано на необходимость трансляции в тип `LPStr`, который является строкой, завершающейся `null`, с однобайтовыми символами ANSI.

На стороне .NET также имеется выбор относительно того, какой тип данных применять. Например, неуправляемые дескрипторы могут отображаться на тип `IntPtr`, `int`, `uint`, `long` или `ulong`.

Большинство неуправляемых дескрипторов инкапсулирует адрес или указатель и потому должно отображаться на `IntPtr` для совместимости с 32- и 64-разрядными операционными системами. Типичным примером может служить `HWND`.

Довольно часто функции Win32 поддерживают целочисленный параметр, который принимает набор констант, определенных в заголовочном файле C++, таком как `WinUser.h`. Вместо определения как простых констант C# их можно представить в виде членов перечисления. Использование перечисления может дать в результате более аккуратный код и увеличить статическую безопасность типов. В разделе "Разделяемая память" далее в главе будет приведен пример.

При установке среды Microsoft Visual Studio удостоверьтесь, что устанавливаете такие заголовочные файлы C++ – даже если в категории C++ не выбрано ничего другого. Именно здесь определены все низкоуровневые константы Win32. Выяснить местонахождение всех заголовочных файлов можно, поискав файлы `*.h` в каталоге программ Visual Studio.

Получение строк из неуправляемого кода обратно в .NET требует проведения некоторых действий по управлению памятью. Маршализатор выполняет такую работу автоматически, если внешний метод объявлен как принимающий объект `StringBuilder`, а не `string`:

```
using System;
using System.Text;
using System.Runtime.InteropServices;

class Test
{
  [DllImport("kernel32.dll")]
  static extern int GetWindowsDirectory (StringBuilder sb, int maxChars);

  static void Main()
  {
    StringBuilder s = new StringBuilder (256);
    GetWindowsDirectory (s, 256);
    Console.WriteLine (s);
  }
}
```

Если вы не уверены, каким образом должен вызываться отдельный метод Win32, тогда поищите пример его вызова в Интернете, указав в качестве строки поиска имя метода и слово `DllImport`. На сайте `http://www.pinvoke.net` стремятся документировать все сигнатуры Win32.

## Маршализация классов и структур

Иногда неуправляемому методу необходимо передавать структуру. Например, метод `GetSystemTime` в API-интерфейсе Win32 определен следующим образом:

```
void GetSystemTime (LPSYSTEMTIME lpSystemTime);
```

Тип `LPSYSTEMTIME` соответствует такой структуре C:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Чтобы вызвать метод `GetSystemTime`, мы должны определить класс или структуру .NET для соответствия показанной выше структуре C:

```
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
class SystemTime
{
  public ushort Year;
  public ushort Month;
  public ushort DayOfWeek;
  public ushort Day;
  public ushort Hour;
  public ushort Minute;
  public ushort Second;
  public ushort Milliseconds;
}
```

Атрибут StructLayout указывает маршализатору, как следует отображать каждое поле на его неуправляемый эквивалент. Член перечисления LayoutKind.Sequential означает, что поля должны выравниваться последовательно по границам *размеров пакета* (объясняется ниже), точно так же как это было бы в структуре C. Имена полей роли не играют; важен только порядок следования полей.

Теперь метод GetSystemTime можно вызывать:

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime (SystemTime t);

static void Main()
{
  SystemTime t = new SystemTime();
  GetSystemTime (t);
  Console.WriteLine (t.Year);
}
```

В языках C и C# поля в объекте располагаются со смещением в *n* байтов, начиная с адреса объекта. Разница в том, что в программе C# среда CLR находит такое смещение с применением маркера поля, а в случае C имена полей компилируются прямо в смещения. Например, в C поле wDay – просто маркер для представления чего-либо, находящегося по адресу экземпляра SystemTime плюс 24 байта.

Для ускорения доступа каждое поле размещается со смещением, кратным размеру поля. Однако используемый множитель ограничен максимумом в *x* байтов, где *x* представляет собой *размер пакета*. В текущей реализации стандартный размер пакета составляет 8 байтов, так что структура, содержащая поле sbyte, за которым следует (8-байтовое) поле long, занимает 16 байтов, и 7 байтов, следующих за sbyte, расходуются впустую. Потери подобного рода можно снизить или вообще устранить, указывая размер пакета через свойство Pack в атрибуте StructLayout: это обеспечивает выравнивание по смещениям, кратным заданному размеру пакета. Таким образом, при размере пакета, равном 1, только что описанная структура будет занимать только 9 байтов. В качестве размера пакета можно указывать 1, 2, 4, 8 или 16 байтов.

Атрибут StructLayout также позволяет задавать явные смещения полей (как показано в разделе "Эмуляция объединения C" далее в главе).

## Маршализация параметров in и out

В предыдущем примере мы реализовали SystemTime в виде класса. Вместо него можно было бы выбрать структуру при условии, что метод GetSystemTime был объявлен с параметром ref или out:

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime (out SystemTime t);
```

В большинстве случаев семантика направленных параметров C# работает одинаково и с внешними методами. Параметры, передаваемые по значению, копируют параметры in, параметры ref в C# копируют параметры in/out, а параметры out в C# копируют параметры out. Тем не менее, существует ряд исключений для типов, которые имеют специальные преобразования. Например, классы массивов и класс StringBuilder требуют копирования при выводе из функции, поэтому они являются in/out. Иногда такое поведение удобно переопределять посредством атрибутов In и Out. Скажем, если массив должен допускать только чтение, то атрибут In указывает, что в функцию поступает только копия массива, но выводиться он из функции не будет:

```
static extern void Foo ( [In] int[] array);
```

# Обратные вызовы из неуправляемого кода

Уровень P/Invoke делает все возможное, чтобы представить естественную модель программирования на обеих сторонах границы, где только возможно обеспечивая отображение между связанными конструкциями. Поскольку в C# можно не только вызывать функции C, но также и осуществлять обратные вызовы из функций C (через указатели на функции), уровень P/Invoke сопоставляет неуправляемые указатели на функции с их ближайшим эквивалентом в C# – делегатами.

В качестве примера рассмотрим метод из библиотеки `User32.dll`, с помощью которого можно перечислять все высокоуровневые оконные дескрипторы:

```
BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

`WNDENUMPROC` – обратный вызов, который последовательно запускается с дескриптором каждого окна (или до тех пор, пока обратный вызов не возвратит `false`). Вот его определение:

```
BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);
```

Для его применения мы объявляем делегат с совпадающей сигнатурой и затем передаем экземпляр этого делегата внешнему методу:

```
using System;
using System.Runtime.InteropServices;
class CallbackFun
{
  delegate bool EnumWindowsCallback (IntPtr hWnd, IntPtr lParam);
  [DllImport("user32.dll")]
  static extern int EnumWindows (EnumWindowsCallback hWnd, IntPtr lParam);
  static bool PrintWindow (IntPtr hWnd, IntPtr lParam)
  {
    Console.WriteLine (hWnd.ToInt64());
    return true;
  }
  static void Main() => EnumWindows (PrintWindow, IntPtr.Zero);
}
```

# Эмуляция объединения C

Каждое поле в структуре получает достаточно места для хранения своих данных. Рассмотрим структуру, содержащую одно поле типа `int` и одно поле типа `char`. Поле `int` вероятно начнется со смещения 0 и гарантированно займет, по меньшей мере, 4 байта. Таким образом, поле `char` начнется со смещения минимум 4. Если по какой-то причине поле `char` начнется со смещения 2, то присваивание значения полю `char` приведет к изменению значения поля `int`. Похоже на хаос, не так ли? Как ни странно, в языке C поддерживается разновидность структуры под названием *объединение*, которая делает именно то, что было описано. Эмулировать объединение в языке C# можно с использованием значения `LayoutKind.Explicit` и атрибута `FieldOffset`.

Выяснение ситуаций, когда объединение может оказаться полезным, иногда затруднительно. Тем не менее, представим, что необходимо воспроизвести ноту на внешнем синтезаторе. Интерфейс Windows Multimedia API предоставляет функцию, которая делает это через протокол MIDI:

```
[DllImport ("winmm.dll")]
public static extern uint midiOutShortMsg (IntPtr handle, uint message);
```

Второй аргумент, message, описывает, какую ноту необходимо воспроизвести. Проблема связана с конструкцией этого 32-битного целого числа без знака: внутренне оно разделено на байты, представляющие канал MIDI, ноту и скорость звучания. Одно из решений предусматривает сдвиг и применение масок через побитовые операции <<, >>, & и | для преобразования таких байтов в и из 32-битного "упакованного" сообщения. Однако намного проще определить структуру с явной компоновкой:

```
[StructLayout (LayoutKind.Explicit)]
public struct NoteMessage
{
  [FieldOffset(0)] public uint PackedMsg;    // Длина 4 байта
  [FieldOffset(0)] public byte Channel;      // FieldOffset также 0
  [FieldOffset(1)] public byte Note;
  [FieldOffset(2)] public byte Velocity;
}
```

Поля Channel, Note и Velocity преднамеренно пересекаются с 32-битным упакованным сообщением, что позволяет осуществлять чтение и запись, используя либо то, либо другое. Для поддержания полей в синхронизированном состоянии никаких дополнительных вычислений не потребуется:

```
NoteMessage n = new NoteMessage();
Console.WriteLine (n.PackedMsg);  // 0
n.Channel = 10;
n.Note = 100;
n.Velocity = 50;
Console.WriteLine (n.PackedMsg);  // 3302410
n.PackedMsg = 3328010;
Console.WriteLine (n.Note);       // 200
```

# Разделяемая память

*Размещенные в памяти файлы*, или *разделяемая память* – это функциональная возможность Windows, которая позволяет множеству процессов на одном компьютере совместно использовать данные без накладных расходов, присущих работе с технологией Remoting или WCF. Разделяемая память является исключительно быстрой и в отличие от каналов предлагает *произвольный* доступ к совместно используемым данным.

В главе 15 было показано, как применять класс MemoryMappedFile для доступа к *размещенным в памяти файлам*; вызов методов Win32 напрямую будет хорошим способом демонстрации уровня P/Invoke.

Функция CreateFileMapping в API-интерфейсе Win32 выделяет разделяемую память. Ей необходимо указать, сколько байтов требуется, а также имя, под которым будет идентифицироваться разделяемая память. Затем другое приложение может подписаться на данную память, вызвав функцию OpenFileMapping с этим именем. Обе функции возвращают *дескриптор*, который можно преобразовать в указатель с помощью вызова функции MapViewOfFile.

Ниже представлен класс, инкапсулирующий доступ к разделяемой памяти:

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;
public sealed class SharedMem : IDisposable
{
  // Здесь мы используем перечисления, потому что они безопаснее констант.
```

```
enum FileProtection : uint   // константы из winnt.h
{
  ReadOnly = 2,
  ReadWrite = 4
}
enum FileRights : uint       // константы из WinBASE.h
{
  Read = 4,
  Write = 2,
  ReadWrite = Read + Write
}
static readonly IntPtr NoFileHandle = new IntPtr (-1);
[DllImport ("kernel32.dll", SetLastError = true)]
static extern IntPtr CreateFileMapping (IntPtr hFile,
                                        int lpAttributes,
                                        FileProtection flProtect,
                                        uint dwMaximumSizeHigh,
                                        uint dwMaximumSizeLow,
                                        string lpName);
[DllImport ("kernel32.dll", SetLastError = true)]
static extern IntPtr OpenFileMapping (FileRights dwDesiredAccess,
                                      bool bInheritHandle,
                                      string lpName);
[DllImport ("kernel32.dll", SetLastError = true)]
static extern IntPtr MapViewOfFile (IntPtr hFileMappingObject,
                                    FileRights dwDesiredAccess,
                                    uint dwFileOffsetHigh,
                                    uint dwFileOffsetLow,
                                    uint dwNumberOfBytesToMap);
[DllImport ("Kernel32.dll", SetLastError = true)]
static extern bool UnmapViewOfFile (IntPtr map);
[DllImport ("kernel32.dll", SetLastError = true)]
static extern int CloseHandle (IntPtr hObject);
IntPtr fileHandle, fileMap;
public IntPtr Root { get { return fileMap; } }
public SharedMem (string name, bool existing, uint sizeInBytes)
{
  if (existing)
    fileHandle = OpenFileMapping (FileRights.ReadWrite, false, name);
  else
    fileHandle = CreateFileMapping (NoFileHandle, 0,
                                    FileProtection.ReadWrite,
                                    0, sizeInBytes, name);
  if (fileHandle == IntPtr.Zero)
    throw new Win32Exception();
  // Получить отображение с возможностью чтения/записи для всего файла.
  fileMap = MapViewOfFile (fileHandle, FileRights.ReadWrite, 0, 0, 0);
  if (fileMap == IntPtr.Zero)
    throw new Win32Exception();
}
```

```
  public void Dispose()
  {
    if (fileMap != IntPtr.Zero) UnmapViewOfFile (fileMap);
    if (fileHandle != IntPtr.Zero) CloseHandle (fileHandle);
    fileMap = fileHandle = IntPtr.Zero;
  }
}
```

В приведенном примере мы указываем `SetLastError=true` на методах `DllImport`, которые используют протокол `SetLastError` для выдачи кодов ошибок. Это обеспечит заполнение исключения `Win32Exception` детальными сведениями об ошибке, когда оно будет сгенерировано. (Вдобавок также появляется возможность запрашивать ошибку явно вызовом метода `Marshal.GetLastWin32Error`.)

Для демонстрации работы класса `SharedMem` понадобится запустить два приложения. Первое из них создает разделяемую память следующим образом:

```
using (SharedMem sm = new SharedMem ("MyShare", false, 1000))
{
  IntPtr root = sm.Root;
  // Появился доступ к разделяемой памяти.

  Console.ReadLine();  // В этот момент мы запускаем второе приложение...
}
```

Второе приложение подписывается на разделяемую память, конструируя объект `SharedMem` с тем же самым именем и передавая значение `true` в качестве аргумента `existing`:

```
using (SharedMem sm = new SharedMem ("MyShare", true, 1000))
{
  IntPtr root = sm.Root;
  // Появился доступ к той же самой разделяемой памяти.
  // ...
}
```

В конечном итоге каждая программа имеет объект `IntPtr` — указатель на одну и ту же неуправляемую память. Теперь два приложения должны каким-то образом читать и записывать в память через имеющийся общий указатель. Один из подходов предполагает построение сериализируемого класса, который инкапсулирует все разделяемые данные, а затем сериализирует (и десериализирует) данные в неуправляемую память с применением класса `UnmanagedMemoryStream`. Однако при наличии большого объема данных такой прием неэффективен. Представьте себе ситуацию, когда класс разделяемой памяти имеет мегабайт полезных данных, но нужно обновить только одно целочисленное значение. Более удачный подход предусматривает определение конструкции разделяемых данных в виде структуры, и затем ее отображение прямо на разделяемую память. Мы обсудим это в следующем разделе.

# Отображение структуры на неуправляемую память

Структура, для которой в атрибуте `StructLayout` указано значение `Sequential` или `Explicit`, может отображаться прямо на неуправляемую память. Рассмотрим показанную ниже структуру:

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
  public int Value;
  public char Letter;
  public fixed float Numbers [50];
}
```

Директива `fixed` позволяет определять массивы типов значения фиксированной длины встроенным образом, что как раз и создает область `unsafe`. Пространство в данной структуре выделяется встроенным образом для 50 чисел с плавающей точкой. В отличие от стандартных массивов C# член `Numbers` — не *ссылка* на массив, а *сам* массив. Если выполнить следующий код:

```
static unsafe void Main() => Console.WriteLine (sizeof (MySharedData));
```

то на консоль выводится результат 208: 50 четырехбайтовых значений `float` плюс 4 байта для поля `Value` типа `int` плюс 2 байта для поля `Letter` типа `char`. Общее количество байтов, равное 206, округляется до 208 из-за того, что значения `float` выравниваются по четырехбайтовым границам (4 байта — размер типа `float`).

Проще всего продемонстрировать использование структуры `MySharedData` в контексте `unsafe` на примере памяти, выделенной в стеке:

```
MySharedData d;
MySharedData* data = &d;  // Получить адрес d

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

или:

```
// Распределить массив в стеке:
MySharedData* data = stackalloc MySharedData[1];

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

Разумеется, мы здесь не демонстрируем ничего такого, чего нельзя было бы достичь в управляемом контексте. Но предположим, что мы хотим хранить экземпляр `MySharedData` в *неуправляемой куче*, т.е. за пределами действия сборщика мусора CLR. Именно в таких случаях указатели становятся по-настоящему полезными:

```
MySharedData* data = (MySharedData*)
  Marshal.AllocHGlobal (sizeof (MySharedData)).ToPointer();

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

Метод `Marshal.AllocHGlobal` выделяет память в неуправляемой куче. Вот как позже освободить ту же самую память:

```
Marshal.FreeHGlobal (new IntPtr (data));
```

(Если забыть об освобождении этой памяти, тогда в результате возникнет хорошо известная утечка памяти.)

Теперь мы будем применять структуру `MySharedData` в сочетании с классом `SharedMem`, написанным в предыдущем разделе. В следующей программе выделяется блок разделяемой памяти, на который затем отображается структура `MySharedData`:

```
static unsafe void Main()
{
  using (SharedMem sm = new SharedMem ("MyShare", false, 1000))
  {
    void* root = sm.Root.ToPointer();
    MySharedData* data = (MySharedData*) root;

    data->Value = 123;
    data->Letter = 'X';
    data->Numbers[10] = 1.45f;
    Console.WriteLine ("Written to shared memory");
                    // Записано в разделяемую память

    Console.ReadLine();

    Console.WriteLine ("Value is " + data->Value);          // Поле Value
    Console.WriteLine ("Letter is " + data->Letter);        // Поле Letter
    Console.WriteLine ("11th Number is " + data->Numbers[10]);
                    // 11-й элемент Numbers
    Console.ReadLine();
  }
}
```

**Вместо SharedMem можно использовать встроенный класс MemoryMappedFile:**

```
using (MemoryMappedFile mmFile =
       MemoryMappedFile.CreateNew ("MyShare", 1000))
using (MemoryMappedViewAccessor accessor =
       mmFile.CreateViewAccessor())
{
  byte* pointer = null;
  accessor.SafeMemoryMappedViewHandle.AcquirePointer
   (ref pointer);
  void* root = pointer;
  ...
}
```

**Ниже представлена вторая программа, которая присоединяется к той же самой разделяемой памяти и читает значения, записанные первой программой. (Она должна быть запущена, пока первая программа ожидает в операторе ReadLine, т.к. после выхода из оператора using объект разделяемой памяти освобождается.)**

```
static unsafe void Main()
{
  using (SharedMem sm = new SharedMem ("MyShare", true, 1000))
  {
    void* root = sm.Root.ToPointer();
    MySharedData* data = (MySharedData*) root;

    Console.WriteLine ("Value is " + data->Value);            // Поле Value
    Console.WriteLine ("Letter is " + data->Letter);          // Поле Letter
    Console.WriteLine ("11th Number is " + data->Numbers[10]);
                    // 11-й элемент Numbers

    // Наша очередь обновлять значения в разделяемой памяти.
    data->Value++;
    data->Letter = '!';
    data->Numbers[10] = 987.5f;
```

```
    Console.WriteLine ("Updated shared memory");
                    // Обновлено в разделяемой памяти
    Console.ReadLine();
  }
}
```

Вывод из обеих программ выглядит так:

```
// Первая программа:

Written to shared memory
Value is 124
Letter is !
11th Number is 987.5

// Вторая программа:

Value is 123
Letter is X
11th Number is 1.45
Updated shared memory
```

Не стоит пугаться указателей: программисты на языке C++ применяют указатели в приложениях повсеместно и способны заставить их работать в любой ситуации. Во всяком случае, большую часть времени. Такой вид использования является сравнительно простым.

Наш пример небезопасен по другой причине. Мы не принимали во внимание проблемы безопасности в отношении потоков (или, выражаясь точнее — безопасности в отношении процессов), которые возникают в ситуации, когда две программы получают доступ к одной и той же памяти одновременно. Чтобы задействовать такой прием в производственном приложении, к полям `Value` и `Letter` структуры `MySharedData` потребуется добавить ключевое слово `volatile`, чтобы предотвратить кеширование этих полей в регистрах центрального процессора. Вдобавок по мере выхода взаимодействия с полями за рамки тривиального почти наверняка придется защищать доступ к ним с помощью межпроцессного объекта `Mutex` — точно так же, как мы бы применяли операторы `lock` для защиты доступа к полям в многопоточной программе. Безопасность к потокам подробно обсуждалась в главе 22.

## `fixed` и `fixed {...}`

Одно из ограничений отображения структур напрямую в память связано с тем, что структуры могут содержать только неуправляемые типы. Если необходимо совместно использовать, например, строковые данные, тогда должен применяться фиксированный массив символов, что означает ручное преобразование в и из типа `string`. Вот как это делать:

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
  ...
  // Выделить пространство для 200 символов (т.е. 400 байтов).
  const int MessageSize = 200;
  fixed char message [MessageSize];

  // Вероятно, данный код имеет смысл поместить во вспомогательный класс:
  public string Message
  {
    get { fixed (char* cp = message) return new string (cp); }
```

```
    set
    {
      fixed (char* cp = message)
      {
        int i = 0;
        for (; i < value.Length && i < MessageSize - 1; i++)
          cp [i] = value [i];
        // Добавить завершающий символ null.
        cp [i] = '\0';
      }
    }
  }
}
```

Понятие вроде ссылки на фиксированный массив отсутствует; взамен вы получаете указатель. При индексации в фиксированном массиве вы на самом деле выполняете арифметические действия над указателями.

С помощью первого использования ключевого слова `fixed` мы выделяем пространство для 200 символов встроенным в структуру образом. Ключевое слово `fixed` имеет другой смысл (что иногда запутывает), когда применяется позже в определении свойства. Оно сообщает среде CLR о необходимости *закрепления* объекта, так что если принимается решение о проведении сборки мусора внутри блока `fixed`, то внутренняя структура не должна перемещаться в рамках кучи (поскольку по ее содержимому будет производиться итерация через прямые указатели в памяти). Глядя на приведенную выше программу, может возникнуть вопрос о том, как вообще структура `MySharedData` может переместиться в памяти, если она располагается не в куче, а в неуправляемой памяти, к которой сборщик мусора не имеет никакого отношения? Однако компилятору ничего не известно о данном факте, и он предполагает, что вы *можете* использовать `MySharedData` в управляемом контексте, поэтому настоятельно требует добавления ключевого слова `fixed`, чтобы сделать код `unsafe` безопасным в управляемых контекстах. И компилятор полностью прав – взгляните, насколько легко поместить структуру `MySharedData` в кучу:

```
object obj = new MySharedData();
```

Результатом будет упакованная структура `MySharedData`, которая находится в куче и может быть перемещена во время сборки мусора.

Рассмотренный пример проиллюстрировал, как строка может быть представлена в структуре, отображаемой на неуправляемую память. Для более сложных типов также доступен вариант применения существующего кода сериализации. Единственное условие – сериализированные данные никогда не должны превышать по длине выделенное для них пространство в структуре, иначе это приведет к непреднамеренному объединению с последующими полями.

# Взаимодействие с COM

С самой первой версии исполняющая среда .NET обладала специальной поддержкой COM, разрешая работать с объектами COM в .NET и наоборот. В версии C# 4.0 поддержка была значительно расширена и усовершенствована в плане как удобства использования, так и развертывания.

## Назначение COM

COM (Component Object Model – модель компонентных объектов) – это двоичный стандарт для API-интерфейсов, выпущенный Microsoft в 1993 году. Мотивацией к созданию COM была необходимость предоставления компонентам возможности взаимодействия друг с другом в независимой от языка и безразличной к версиям манере. До появления COM подход, применяемый в Windows, заключался в опубликовании *динамически подключаемых библиотек* (Dynamic Link Library – DLL), которые объявляли структуры и функции с использованием языка программирования C. Такой подход не только зависел от языка, но также был достаточно хрупким. Спецификация типа в библиотеке подобного рода неотделима от его реализации: даже добавление к структуре нового поля разрушало ее спецификацию.

Элегантность COM заключалась в отделении спецификации типа от его реализации через конструкцию, известную как *интерфейс COM*. Технология COM также позволила вызывать методы на *объектах*, поддерживающих состояние – не ограничиваясь простыми вызовами процедур.

В определенном смысле модель программирования для .NET является эволюцией принципов программирования для COM: платформа .NET также упрощает разработку на множестве языков и позволяет двоичным компонентам развиваться, не нарушая работу приложений, которые от них зависят.

## Основы системы типов COM

Система типов COM вращается вокруг интерфейсов. Интерфейс COM довольно похож на интерфейс .NET, но получил большее распространение из-за того, что тип COM открывает свою функциональность *только* через интерфейс. Например, вот как мы могли бы объявить тип в мире .NET:

```
public class Foo
{
  public string Test() => "Hello, world";
}
```

Потребители типа `Foo` могут применять метод `Test` напрямую. И если позже будет изменена *реализация* метода `Test`, то повторная компиляция вызывающих сборок не потребуется. В таком отношении платформа .NET отделяет интерфейс от реализации, не делая интерфейсы обязательными. Можно было бы даже добавить перегруженную версию метода `Test`, не нарушив работу вызывающих компонентов:

```
public string Test (string s) => "Hello, world " + s;
```

В мире COM для достижения такого же уровня развязки класс `Foo` открывает свою функциональность через интерфейс. Таким образом, в библиотеке типов `Foo` будет присутствовать интерфейс, подобный представленному ниже:

```
public interface IFoo { string Test(); }
```

(Мы иллюстрировали сказанное, показав интерфейс C# – не интерфейс COM. Тем не менее, принцип остается тем же самым, хотя связующий код отличается.)

Вызывающие компоненты будут затем взаимодействовать с `IFoo`, а не с `Foo`.

Когда дело доходит до добавления перегруженной версии метода `Test`, ситуация с технологией COM оказывается более сложной, чем с .NET. Во-первых, мы хотели бы

избежать модификации интерфейса IFoo, т.к. это нарушило бы двоичную совместимость с предыдущей версией (один из принципов COM заключается в том, что интерфейсы после опубликования являются *неизменяемыми*). Во-вторых, технология COM не поддерживает перегрузку методов. Решение состоит в том, чтобы обеспечить реализацию классом Foo *другого интерфейса*:

```
public interface IFoo2 { string Test (string s); }
```

(И снова для придания знакомого вида мы представили его в виде интерфейса .NET.)

Поддержка множества интерфейсов играет ключевую роль в возможности создания версий библиотек COM.

### IUnknown и IDispatch

Все интерфейсы COM идентифицируются с помощью глобально уникального идентификатора (GUID).

Корневым интерфейсом в COM является IUnknown; его обязаны реализовывать все объекты COM. Он имеет три метода:

- AddRef
- Release
- QueryInterface

Методы AddRef и Release предназначены для управления временем жизни, поскольку в COM используется подсчет ссылок, а не автоматическая сборка мусора (технология COM была спроектирована для работы с неуправляемым кодом, в котором автоматическая сборка мусора неосуществима). Метод QueryInterface возвращает ссылку на объект, который поддерживает данный интерфейс, если он способен делать это.

Чтобы стало возможным динамическое программирование (например, написание сценариев и автоматизация), объект COM может также реализовывать интерфейс IDispatch. В результате у динамических языков вроде VBScript появляется возможность обращаться к объектам COM с применением позднего связывания – почти как с помощью dynamic в C# (хотя только для простых вызовов).

## Обращение к компоненту COM из C#

Наличие встроенной в CLR поддержки для COM означает, что работать напрямую с интерфейсами IUnknown и IDispatch не придется. Взамен вы имеете дело с объектами CLR, а исполняющая среда маршализирует обращения к миру COM через *вызываемые оболочки времени выполнения* (Runtime-Callable Wrapper – RCW). Исполняющая среда также отвечает за управление временем жизни, вызывая методы AddRef и Release (когда объект .NET финализируется), и заботится о преобразованиях примитивных типов между двумя мирами. Преобразование типов гарантирует, что каждая сторона видит, например, целочисленные и строковые типы в знакомых ей формах.

Кроме того, необходим какой-нибудь способ доступа к оболочкам RCW в статически типизированной манере. Такая работа выполняется *типами взаимодействия с COM*. Типы взаимодействия с COM – это автоматически сгенерированные прокси-типы, которые открывают доступ к члену .NET для каждого члена COM. Инструмент импорта библиотек типов (tlbimp.exe) генерирует типы взаимодействия с COM в командной строке на основе выбранной библиотеки COM и компилирует их в *сборку взаимодействия с COM*.

Если компонент COM реализует сразу несколько интерфейсов, тогда инструмент `tlbimp.exe` генерирует одиночный тип, который содержит объединение членов из всех интерфейсов.

Сборку взаимодействия с COM можно создать в Visual Studio, открыв диалоговое окно Add Reference (Добавить ссылку) и выбрав нужную библиотеку на вкладке COM. Например, при наличии установленной программы Microsoft Excel добавление ссылки на библиотеку Microsoft Excel Interop Library позволяет взаимодействовать с классами COM для Excel. Ниже приведен код, который создает и отображает рабочую книгу, а затем заполняет в ней ячейку:

```
using System;
using Excel = Microsoft.Office.Interop.Excel;

class Program
{
  static void Main()
  {
    var excel = new Excel.Application();
    excel.Visible = true;
    Excel.Workbook workBook = excel.Workbooks.Add();
    excel.Cells [1, 1].Font.FontStyle = "Bold";
    excel.Cells [1, 1].Value2 = "Hello World";
    workBook.SaveAs (@"d:\temp.xlsx");
  }
}
```

Класс `Excel.Application` — это тип взаимодействия с COM, чьим типом времени выполнения является RCW. Когда мы обращаемся к свойствам `Workbooks` и `Cells`, то получаем еще больше типов взаимодействия.

Благодаря введенным в версии C# 4.0 улучшениям, связанным с COM, код получился довольно простым. Без таких улучшений метод `Main` выглядел бы следующим образом:

```
var missing = System.Reflection.Missing.Value;

var excel = new Excel.Application();
excel.Visible = true;
Excel.Workbook workBook = excel.Workbooks.Add (missing);
var range = (Excel.Range) excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
range.Value2 = "Hello world";

workBook.SaveAs (@"d:\temp.xlsx", missing, missing, missing, missing,
  missing, Excel.XlSaveAsAccessMode.xlNoChange, missing, missing,
  missing, missing, missing);
```

Давайте теперь посмотрим, что собой представляют языковые улучшения и как они помогают при программировании для COM.

## Необязательные параметры и именованные аргументы

Поскольку API-интерфейсы COM не поддерживают перегрузку функций, очень часто приходится иметь дело с функциями, принимающими многочисленные параметры, часть которых являются необязательными. Например, вот как можно вызвать метод `Save` рабочей книги Excel:

```
var missing = System.Reflection.Missing.Value;
workBook.SaveAs (@"d:\temp.xlsx", missing, missing, missing, missing,
  missing, Excel.XlSaveAsAccessMode.xlNoChange, missing, missing,
  missing, missing, missing);
```

Хорошая новость в том, что поддержка необязательных параметров в C# осведомлена о COM, поэтому можно поступать просто так:

```
workBook.SaveAs (@"d:\temp.xlsx");
```

(Как объяснялось в главе 3, необязательные параметры "расширяются" компилятором в полную форму.)

Именованные аргументы позволяют указывать дополнительные аргументы независимо от их позиций:

```
workBook.SaveAs (@"c:\test.xlsx", Password:"foo");
```

## Неявные параметры `ref`

Некоторые API-интерфейсы COM (в частности, Microsoft Word) открывают доступ к функциям, которые объявляют *каждый* параметр как передаваемый по ссылке вне зависимости от того, модифицирует функция его значение или нет. Причина — выигрыш в производительности из-за отсутствия необходимости копировать значения аргументов (хотя *фактический* выигрыш в производительности незначителен).

Исторически сложилось так, что вызов методов подобного рода в коде C# был затруднен, поскольку приходилось указывать ключевое слово `ref` для каждого аргумента, а это предотвращало использование необязательных параметров. Например, для открытия документа Word раньше нужно было поступать следующим образом:

```
object filename = "foo.doc";
object notUsed1 = Missing.Value;
object notUsed2 = Missing.Value;
object notUsed3 = Missing.Value;
...
Open (ref filename, ref notUsed1, ref notUsed2, ref notUsed3, ...);
```

Тем не менее, начиная с версии C# 4.0, модификатор `ref` в вызовах функций COM можно опускать, делая возможным применение необязательных параметров:

```
word.Open ("foo.doc");
```

Однако следует помнить о том, что если вызываемый метод COM действительно изменит значение аргумента, то никакой ошибки не возникнет — ни на этапе компиляции, ни во время выполнения.

## Индексаторы

Возможность не указывать модификатор `ref` дает еще одно преимущество: индексаторы COM с параметрами `ref` становятся доступными через обычный синтаксис индексаторов C#. В противном случае это было бы запрещено, т.к. параметры `ref`/`out` не поддерживаются индексаторами C# (несколько неуклюжим способом обхода указанной проблемы в старых версиях C# был вызов опорных методов, таких как `get_`*XXX* и `set_`*XXX*; данный прием по-прежнему законен для достижения обратной совместимости).

В версии C# 4.0 взаимодействие с индексаторами было подвергнуто дальнейшим усовершенствованиям, что позволило обращаться к свойствам COM, которые принимают аргументы.

В следующем примере Foo является свойством, принимающим целочисленный аргумент:

```
myComObject.Foo [123] = "Hello";
```

Самостоятельное написание таких свойств в C# все еще запрещено: тип может открывать доступ к индексатору только на самом себе ("стандартный" индексатор). Таким образом, если необходимо написать код C#, который сделал бы предыдущий оператор законным, то свойство Foo должно было бы возвращать другой тип, открывающий доступ к (стандартному) индексатору.

## Динамическое связывание

Есть два способа, которыми динамическое связывание может помочь при обращении к компонентам COM. Первый из них касается доступа к компоненту COM без типа взаимодействия COM. Понадобится вызвать метод Type.GetTypeFromProgID с именем компонента COM для получения экземпляра COM, после чего использовать динамическое связывание для вызова методов данного экземпляра. Естественно, средство IntelliSense здесь недоступно, и проверки на этапе компиляции невозможны:

```
Type excelAppType = Type.GetTypeFromProgID ("Excel.Application", true);
dynamic excel = Activator.CreateInstance (excelAppType);
excel.Visible = true;
dynamic wb = excel.Workbooks.Add();
excel.Cells [1, 1].Value2 = "foo";
```

(Аналогичной цели можно достичь и намного более запутанным путем, применив рефлексию вместо динамического связывания.)

Вариацией на эту тему является обращение к компоненту COM, который поддерживает *только* интерфейс IDispatch. Тем не менее, такие компоненты встречаются довольно редко.

Динамическое связывание также может быть удобным (в меньшей степени) при работе с COM-типом variant. По причинам, обусловленным скорее неудачным проектным решением, нежели необходимостью, функции API-интерфейса COM часто буквально усыпаны данным типом, который является грубым эквивалентом типа object в .NET. Если вы включите параметр Embed Interop Types (Внедрять типы взаимодействия) в своем проекте (как более подробно объясняется далее), тогда исполняющая среда будет отображать variant на dynamic вместо отображения variant на object, устраняя необходимость в приведениях. Например, законно было бы сделать так:

```
excel.Cells [1, 1].Font.FontStyle = "Bold";
```

вместо:

```
var range = (Excel.Range) excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

Недостаток такого способа работы связан с утерей возможности автозавершения, поэтому вы должны знать, что свойство по имени Font точно существует. По указанной причине обычно проще *динамически* присваивать результат известному типу взаимодействия:

```
Excel.Range range = excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

Как видите, код не намного короче, чем при подходе в старом стиле.

Отображение variant на dynamic принято по умолчанию в Visual Studio 2010 и последующих версиях и управляется включением параметра Embed Interop Types (Внедрять типы взаимодействия) для ссылки.

# Внедрение типов взаимодействия

Ранее упоминалось о том, что C# обычно обращается к компонентам COM через типы взаимодействия, которые генерируются путем запуска инструмента tlbimp.exe (напрямую или через Visual Studio).

Исторически сложилось так, что единственной возможностью была *ссылка* на сборки взаимодействия, как в случае любых других сборок. Дело могло быть хлопотным, потому что сборки взаимодействия для сложных компонентов COM зачастую оказываются довольно большими. Скажем, крошечный дополнительный модуль для Microsoft Word требует сборки взаимодействия, которая на порядки больше его самого.

Начиная с версии C# 4.0, в дополнение к *ссылке* на сборку взаимодействия появилась возможность *связываться* с ней. В таком случае компилятор анализирует сборку на предмет типов и членов, действительно используемых в приложении. Затем он внедряет определения для таких типов и членов прямо в приложение. Это означает, что переживать по поводу раздувания кода не придется, поскольку в приложение будут включены только те интерфейсы COM, которые действительно применяются.

В Visual Studio 2010 и последующих версиях связывание со сборками взаимодействия для ссылок COM устанавливается по умолчанию. Если нужно *отключить* его, выберите ссылку в проводнике решения, откройте окно ее свойств и установите параметр Embed Interop Types (Внедрять типы взаимодействия) в False.

Чтобы включить связывание со сборками взаимодействия в компиляторе командной строки, запускайте csc с ключом /link вместо /reference (или /L вместо /R).

## Эквивалентность типов

Среда CLR 4.0 и ее более новые версии поддерживают *эквивалентность типов* для связанных типов взаимодействия. Следовательно, когда две сборки связываются с каким-то типом взаимодействия, то такие типы будут считаться эквивалентными, если они являются оболочками для одного и того же типа COM. Сказанное справедливо, даже когда сборки взаимодействия, с которыми они связаны, были сгенерированы независимо друг от друга.

Эквивалентность типов полагается на атрибут TypeIdentifierAttribute из пространства имен System.Runtime.InteropServices. Компилятор автоматически применяет его во время связывания со сборками взаимодействия. Затем типы COM считаются эквивалентными, если они имеют одинаковые GUID.

Эквивалентность типов устраняет потребность в основных сборках взаимодействия.

# Основные сборки взаимодействия

До выхода версии C# 4.0 не существовало ни связывания со сборками взаимодействия, ни эквивалентности типов. Это создавало проблему в ситуации, когда два разработчика запускали инструмент tlbimp.exe на том же самом компоненте COM:

в итоге они получали несовместимые сборки взаимодействия, что затрудняло саму возможность взаимодействия. Обходной путь заключался в том, что авторы каждой библиотеки COM выпускали официальную версию сборки взаимодействия, которая называлась *основной сборкой взаимодействия* (Primary Interop Assembly – PIA). Сборки PIA по-прежнему распространены, главным образом из-за наличия большого объема унаследованного кода.

Сборки PIA являются неудачным решением по описанным ниже причинам.

- *Сборки PIA использовались не всегда.* Поскольку запускать инструмент импорта библиотек типов мог кто угодно, часто так и поступали вместо применения официальной версии. В некоторых случаях другого выбора и не было, т.к. авторы интересующей библиотеки COM просто не публиковали для нее сборку PIA.
- *Сборки PIA требуют регистрации.* Сборки PIA требуют регистрации в GAC. Это бремя возлагается даже на разработчиков, пишущих простые дополнения для какого-то компонента COM.
- *Сборки PIA увеличивают размеры развертывания.* Сборки PIA служат примером ранее описанной проблемы раздувания кода в сборках взаимодействия. В частности, команда разработчиков Microsoft Office решила не развертывать сборки PIA со своим продуктом.

# Открытие объектов C# для COM

Существует также возможность написания классов C#, которые могут потребляться в мире COM. Среда CLR делает это возможным через прокси под названием *вызываемая оболочка COM* (COM-callable wrapper – CCW). Оболочка CCW маршализирует типы между двумя мирами (подобно RCW), а также реализует интерфейс `IUnknown` (и дополнительно `IDispatch`), как того требует протокол COM. Временем жизни оболочки CCW управляет сторона COM через подсчет ссылок (а не посредством сборщика мусора CLR).

Для COM можно делать доступным любой открытый класс. Единственным требованием является определение атрибута сборки, который назначает GUID с целью идентификации библиотеки типов COM:

```
[assembly: Guid ("...")]   // Уникальный GUID для библиотеки типов COM
```

По умолчанию все открытые типы будут видимыми для потребителей на стороне COM. Однако определенные типы можно сделать невидимыми, применив к ним атрибут `[ComVisible(false)]`. Если нужно сделать все типы невидимыми по умолчанию, тогда к сборке следует применить атрибут `[ComVisible(false)]`, а к типам, которые должны быть видимыми – атрибут `[ComVisible(true)]`.

Финальный шаг заключается в запуске инструмента `tlbexp.exe`:

```
tlbexp.exe myLibrary.dll
```

В результате генерируется файл библиотеки типов COM (`.tlb`), который можно регистрировать и использовать в приложениях COM. Интерфейсы COM для сопоставления с классами, видимыми из COM, генерируются автоматически.

# 26

# Регулярные выражения

Язык регулярных выражений распознает символьные образцы. Типы .NET, поддерживающие регулярные выражения, основаны на регулярных выражениях Perl 5 и обеспечивают функциональность как поиска, так и поиска/замены.

Регулярные выражения используются для решения следующих задач:

- проверка текстового ввода, такого как пароли и телефонные номера (для такой цели в ASP.NET предлагается элемент управления `RegularExpressionValidator`);
- преобразование текстовых данных в более структурированные формы (например, извлечение данных из HTML-страницы с целью их сохранения в базе данных);
- замена образцов текста в документе (например, только целых слов).

Настоящая глава разделена на концептуальные разделы, обучающие основам регулярных выражений в .NET, и справочные разделы, в которых приводится описание языка регулярных выражений.

Все типы для работы с регулярными выражениями определены в пространстве имен `System.Text.RegularExpressions`.

Более подробные сведения о регулярных выражениях можно найти на веб-сайте `http://regular-expressions.info`, который содержит удобный онлайновый справочник с множеством примеров, и в книге Джеффри Фридла *Mastering Regular Expressions* (`http://oreilly.com/catalog/9781565922570`), бесценной для серьезных программистов.

Все примеры, приведенные в главе, можно загрузить вместе с LINQPad. Доступна также интерактивная утилита под названием Expresso (`http://www.ultrapico.com`), которая помогает строить и визуализировать регулярные выражения и содержит собственную библиотеку выражений.

## Основы регулярных выражений

Одной из наиболее распространенных операций регулярных выражений является *квантификатор*. Операция `?` – это квантификатор, который соответствует предшествующему элементу 0 или 1 раз. Другими словами, `?` означает *необязательный*.

Элемент представляет собой либо одиночный символ, либо сложную структуру символов в квадратных скобках. Например, регулярное выражение "colou?r" соответствует color и colour, но не colouur:

```
Console.WriteLine (Regex.Match ("color",   @"colou?r").Success);   // True
Console.WriteLine (Regex.Match ("colour",  @"colou?r").Success);   // True
Console.WriteLine (Regex.Match ("colouur", @"colou?r").Success);   // False
```

Метод Regex.Match выполняет поиск внутри большой строки. Возвращаемый им объект имеет свойства для позиции (Index) и длины (Length) совпадения, а также свойство для действительного значения (Value) совпадения:

```
Match m = Regex.Match ("any colour you like", @"colou?r");

Console.WriteLine (m.Success);     // True
Console.WriteLine (m.Index);       // 4
Console.WriteLine (m.Length);      // 6
Console.WriteLine (m.Value);       // colour
Console.WriteLine (m.ToString()); // colour
```

Метод Regex.Match можно воспринимать как более мощную версию метода IndexOf типа string. Разница в том, что он ищет совпадение с *образцом*, а не с литеральной строкой.

Метод IsMatch – сокращение для вызова метода Match с последующей проверкой свойства Success.

Механизм регулярных выражений по умолчанию работает слева направо, поэтому возвращается только самое левое соответствие. Для возвращения дополнительных совпадений можно применять метод NextMatch:

```
Match m1 = Regex.Match ("One color? There are two colours in my head!",
                        @"colou?rs?");
Match m2 = m1.NextMatch();
Console.WriteLine (m1);            // color
Console.WriteLine (m2);            // colours
```

Метод Matches возвращает все совпадения в виде массива. Предыдущий пример можно переписать, как показано ниже:

```
foreach (Match m in Regex.Matches
        ("One color? There are two colours in my head!", @"colou?rs?"))
  Console.WriteLine (m);
```

Еще одной распространенной операцией регулярных выражений является *перестановка*, обозначаемая вертикальной чертой, т.е. |. Перестановка выражает альтернативы. Следующий код дает совпадения для Jen, Jenny и Jennifer:

```
Console.WriteLine (Regex.IsMatch ("Jenny", "Jen(ny|nifer)?")); // True
```

Скобки вокруг перестановки отделяют альтернативы от остальной части выражения.

Начиная с версии .NET Framework 4.5, при поиске совпадений с регулярными выражениями можно указывать тайм-аут. Если операция поиска совпадения занимает больше времени, чем заданное в объекте TimeSpan, тогда генерируется исключение RegexMatchTimeoutException. Прием может быть полезен, когда программа обрабатывает произвольные регулярные выражения (например, в диалоговом окне расширенного поиска), потому что он предотвращает бесконечное зацикливание неправильно сформированных регулярных выражений.

## Скомпилированные регулярные выражения

В некоторых рассмотренных ранее примерах мы вызывали статический метод RegEx многократно с одним и тем же образцом. В таких случаях альтернативным подходом является создание объекта Regex с этим образцом и флагом RegexOptions.Compiled, а затем вызов методов экземпляра:

```
Regex r = new Regex (@"sausages?" , RegexOptions.Compiled);
Console.WriteLine (r.Match ("sausage"));    // sausage
Console.WriteLine (r.Match ("sausages"));   // sausages
```

Флаг RegexOptions.Compiled инструктирует экземпляр RegEx о том, что должна использоваться облегченная генерация кода (DynamicMethod в Reflection.Emit) для динамического построения и компиляции кода, настроенного на это конкретное выражение. В результате обеспечивается более быстрое сопоставление за счет затрат на первоначальную компиляцию.

Экземпляр Regex является неизменяемым.

Механизм регулярных выражений характеризуется высокой скоростью. Даже без компиляции нахождение простого совпадения требует менее микросекунды.

## RegexOptions

Перечисление флагов RegexOptions позволяет настраивать поведение сопоставления. Распространенное применение RegexOptions связано с выполнением поиска, нечувствительного к регистру символов:

```
Console.WriteLine (Regex.Match ("a", "A", RegexOptions.IgnoreCase));  // a
```

Это задействует правила для эквивалентности регистров символов текущей культуры. Флаг CultureInvariant позволяет затребовать инвариантную культуру:

```
Console.WriteLine (Regex.Match ("a", "A", RegexOptions.IgnoreCase
                                 | RegexOptions.CultureInvariant));
```

Большинство флагов RegexOptions могут также активизироваться внутри самого регулярного выражения с использованием однобуквенного кода:

```
Console.WriteLine (Regex.Match ("a", @"(?i)A"));          // a
```

Действие флагов можно включать и отключать на протяжении всего выражения:

```
Console.WriteLine (Regex.Match ("AAAa", @"(?i)a(?-i)a")); // Aa
```

Еще одним полезным флагом является IgnorePatternWhitespace или (?x). Он позволяет вставлять пробельные символы, чтобы улучшить читабельность регулярного выражения – без трактовки таких символов литеральным образом.

В табл. 26.1 перечислены все значения RegExOptions вместе с их однобуквенными кодами.

## Отмена символов

Регулярные выражения имеют следующие метасимволы, которые трактуются специальным образом, отличающимся от их литерального смысла:

```
\ * + ? | { [ () ^ $ . #
```

**Таблица 26.1. Параметры регулярных выражений**

| Значение перечисления | Однобуквенный код | Описание |
|---|---|---|
| None | | |
| IgnoreCase | i | Игнорировать регистр символов (по умолчанию регулярные выражения чувствительны к регистру символов) |
| Multiline | m | Изменить ^ и $ так, чтобы они соответствовали началу/концу строки текста, а не началу/концу всей строки регулярного выражения |
| ExplicitCapture | n | Захватывать только явно именованные или явно нумерованные группы (как описано в разделе "Группы" далее в главе) |
| Compiled | | Инициировать компиляцию регулярного выражения в IL (см. раздел "Скомпилированные регулярные выражения" ранее в главе) |
| Singleline | s | Сделать точку (.) соответствующей любому символу (вместо соответствия любому символу кроме \n) |
| IgnorePatternWhitespace | x | Устранить из образца неотмененные пробельные символы |
| RightToLeft | r | Выполнять поиск справа налево; указывать посреди операции не допускается |
| ECMAScript | | Обеспечить совместимость с ECMA (по умолчанию реализация не совместима с ECMA) |
| CultureInvariant | | Отключить поведение, специфичное для культуры, при сравнении строк |

Чтобы применить метасимвол литерально, его потребуется предварить обратной косой чертой. В следующем примере мы отменяем символ ? для сопоставления со строкой "what?":

```
Console.WriteLine (Regex.Match ("what?", @"what\?")); // what? (правильно)
Console.WriteLine (Regex.Match ("what?", @"what?")); // what  (неправильно)
```

Если символ находится внутри *набора* (в квадратных скобках), то данное правило не действует, и метасимволы интерпретируются литеральным образом. Наборы обсуждаются в следующем разделе.

Методы Escape и Unescape класса Regex преобразуют строку, содержащую метасимволы регулярных выражений, путем замены их отмененными эквивалентами и наоборот. Например:

```
Console.WriteLine (Regex.Escape   (@"?"));          // \?
Console.WriteLine (Regex.Unescape (@"\?"));         // ?>
```

Все строки регулярных выражений в настоящей главе представлены с помощью литерала @ из C#. Так сделано для того, чтобы обойти механизм отмены языка C#, в котором также используется обратная косая черта. Без символа @ литеральная обратная косая черта потребовала бы указания четырех таких символов:

```
Console.WriteLine (Regex.Match ("\\", "\\\\"));        // \
```

Если не включена опция (?x), тогда пробелы в регулярных выражениях трактуются литеральным образом:

```
Console.Write (Regex.IsMatch ("hello world", @"hello world")); // True
```

## Наборы символов

Наборы символов действуют в качестве групповых символов для отдельного множества символов.

| Выражение | Описание | Инверсия ("не") |
|---|---|---|
| [abcdef] | Соответствие одиночному символу в списке | [^abcdef] |
| [a-f] | Соответствие одиночному символу в *диапазоне* | [^a-f] |
| \d | Соответствие десятичной цифре. То же самое, что и [0-9] | \D |
| \w | Соответствие символу, который допустим в *словах* (по умолчанию варьируется согласно CultureInfo.CurrentCulture; например, в английском языке это то же самое, что и [a-zA-Z_0-9]) | \W |
| \s | Соответствие пробельному символу. То же самое, что и [\n\r\t\f\v ] | \S |
| \p{*категория*} | Соответствие символу в указанной *категории* (табл. 26.6) | \P |
| . | (Стандартный режим.) Соответствие любому символу кроме \n | \n |
| . | (Режим SingleLine.) Соответствие любому символу | \n |

Для соответствия в точности одному символу из набора поместите набор символов в квадратные скобки:

```
Console.Write (Regex.Matches ("That is that.", "[Tt]hat").Count); // 2
```

Для соответствия любому символу, *исключая* перечисленные в наборе, поместите набор в квадратные скобки и укажите ^ перед первым символом набора:

```
Console.Write (Regex.Match ("quiz qwerty", "q[^aeiou]").Index);   // 5
```

С помощью дефиса можно задавать диапазон символов. Следующее выражение соответствует шахматному ходу:

```
Console.Write (Regex.Match ("b1-c4", @"[a-h]\d-[a-h]\d").Success); // True
```

\d указывает цифровой символ, поэтому \d будет соответствовать любой цифре.

\D соответствует любому нецифровому символу.

\w указывает символ, допустимый в словах, что включает буквы, цифры и подчеркивание.

\W соответствует любому символу, наличие которого в словах не допускается. Это также работает ожидаемым образом и для неанглийских букв, таких как кириллица.

. соответствует любому символу кроме \n (но разрешает \r).

\p соответствует символу в указанной категории, такой как {Lu} для буквы верхнего регистра или {P} для знака пунктуации (список категорий будет приведен в справочном разделе далее в главе):

```
Console.Write (Regex.IsMatch ("Yes, please", @"\p{P}"));  // True
```

Мы приведем больше случаев применения \d, \w и ., когда будем комбинировать их с *квантификаторами*.

# Квантификаторы

Квантификаторы обеспечивают соответствие элементу указанное количество раз.

| Квантификатор | Описание |
|---|---|
| * | Ноль или больше совпадений |
| + | Одно или больше совпадений |
| ? | Ноль или одно совпадение |
| {*n*} | В точности *n* совпадений |
| {*n*,} | По меньшей мере, *n* совпадений |
| {*n*,*m*} | Количество совпадений между *n* и *m* |

Квантификатор * обеспечивает соответствие предшествующего символа или группы ноль или более раз. Следующее выражение соответствует имени файла cv.doc, а также любым версиям имени с числами (например, cv2.doc, cv15.doc):

```
Console.Write (Regex.Match ("cv15.doc", @"cv\d*\.doc").Success);    // True
```

Обратите внимание, что мы должны отменить символ точки в расширении файла с помощью обратной косой черты.

Показанное ниже выражение допускает наличие любых символов между cv и .doc и эквивалентно команде dir cv*.doc:

```
Console.Write (Regex.Match ("cvjoint.doc", @"cv.*\.doc").Success);  // True
```

Квантификатор + обеспечивает соответствие предшествующего символа или группы один или более раз, например:

```
Console.Write (Regex.Matches ("slow! yeah slooow!", "slo+w").Count);// 2
```

Квантификатор { } обеспечивает соответствие указанному количеству (или диапазону) повторений. Следующее выражение выводит показания артериального давления:

```
Regex bp = new Regex (@"\d{2,3}/\d{2,3}");
Console.WriteLine (bp.Match ("It used to be 160/110"));    // 160/110
Console.WriteLine (bp.Match ("Now it's only 115/75"));     // 115/75
```

## Жадные и ленивые квантификаторы

По умолчанию квантификаторы являются *жадными* как противоположность *ленивым* квантификаторам. Жадный квантификатор повторяется настолько *много* раз, сколько может, прежде чем продолжить. Ленивый квантификаторы повторяется настолько *мало* раз, сколько может, прежде чем продолжить. Для того чтобы сделать любой квантификатор ленивым, его необходимо снабдить суффиксом в виде символа ?.

Чтобы проиллюстрировать разницу, рассмотрим следующий фрагмент HTML-разметки:

```
string html = "<i>By default</i> quantifiers are <i>greedy</i> creatures";
```

Предположим, что нужно извлечь две фразы, выделенные курсивом. Если мы запустим следующий код:

```
foreach (Match m in Regex.Matches (html, @"<i>.*</i>"))
  Console.WriteLine (m);
```

то результатом будет не два, а *одно* совпадение:

```
<i>By default</i> quantifiers are <i>greedy</i>
```

Проблема в том, что квантификатор * жадным образом повторяется настолько много раз, сколько может, перед обнаружением соответствия </i>. Таким образом, он поглощает первое вхождение </i>, останавливаясь только на финальном вхождении </i> (*последняя точка*, где все еще обеспечивается совпадение).

Если сделать квантификатор ленивым:

```
foreach (Match m in Regex.Matches (html, @"<i>.*?</i>"))
  Console.WriteLine (m);
```

тогда он остановится в *первой* точке, после которой остаток выражения может дать совпадение. Вот результат:

```
<i>By default</i>
<i>greedy</i>
```

# Утверждения нулевой ширины

Язык регулярных выражений позволяет размещать условия, которые должны удовлетворяться *перед* или *после* совпадения, через *просмотр назад*, *просмотр вперед*, *привязки* и *границы слов*. Все вместе они называется утверждениями *нулевой ширины*, потому что они не увеличивают ширину (или длину) самого совпадения.

## Просмотр вперед и просмотр назад

Конструкция `(?=expr)` проверяет, соответствует ли следующий за ней текст выражению `expr`, не включая `expr` в результат. Это называется *положительным просмотром вперед*. В приведенном ниже примере мы ищем число, за которым расположено слово `miles`:

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles)"));
```

```
ВЫВОД: 25
```

Обратите внимание, что слово `miles` не возвращается как часть результата, хотя оно требовалось, чтобы *удовлетворить* условие совпадения.

После успешного просмотра вперед поиск совпадения продолжается, как если бы предварительный просмотр никогда не выполнялся. Таким образом, если добавить к выражению конструкцию `.*`, как показано ниже:

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles).*"));
```

то результатом будет `25 miles more`.

Просмотр вперед может быть полезен для навязывания правил выбора сильных паролей. Предположим, что пароль должен иметь длину не менее шести символов и содержать, по крайней мере, одну цифру.

С помощью просмотра задачу можно решить следующим образом:

```
string password = "...";
bool ok = Regex.IsMatch (password, @"(?=.*\d).{6,}");
```

Здесь сначала осуществляется *просмотр вперед*, чтобы удостовериться в наличии цифры где-нибудь в строке. Если цифра обнаружена, тогда происходит возврат к позиции перед началом предварительного просмотра и производится проверка соответствия шести или более символам. (В разделе "Рецептурный справочник по регулярным выражениям" далее в главе мы приводим более существенный пример проверки паролей.)

Противоположностью является конструкция *отрицательного просмотра вперед*, т.е. `(?!expr)`. Она требует, чтобы совпадение *не* следовало за выражением `expr`. Приведенное далее выражение соответствует `good`, если только позже в строке не встречается `however` или `but`:

```
string regex = "(?i)good(?!.*(however|but))";
Console.WriteLine (Regex.IsMatch ("Good work! But...", regex));  // False
Console.WriteLine (Regex.IsMatch ("Good work! Thanks!", regex));  // True
```

Конструкция `(?<=expr)` обозначает *положительный просмотр назад* и требует, чтобы совпадению *предшествовало* указанное выражение. Противоположная конструкция, `(?<!expr)`, обозначает *отрицательный просмотр назад* и требует, чтобы совпадению *не предшествовало* указанное выражение. Например, следующее выражение соответствует `good`, если только `however` не встречалось *ранее* в строке:

```
string regex = "(?i)(?<!however.*)good";
Console.WriteLine (Regex.IsMatch ("However good, we...", regex));  // False
Console.WriteLine (Regex.IsMatch ("Very good, thanks!", regex));  // True
```

Приведенные примеры можно было бы усовершенствовать, добавив *утверждения границ слов*, которые вскоре будут описаны.

## Привязки

Привязки `^` и `$` соответствуют конкретной *позиции*. По умолчанию:

- `^` соответствует началу строки;
- `$` соответствует концу строки.

В зависимости от контекста символ `^` означает *привязку* или *отрицание класса символов*. В зависимости от контекста символ `$` означает *привязку* или *маркер группы замены*.

Например:

```
Console.WriteLine (Regex.Match ("Not now", "^[Nn]o"));  // No
Console.WriteLine (Regex.Match ("f = 0.2F", "[Ff]$"));  // F
```

Если указать `RegexOptions.Multiline` или включить в выражение конструкцию `(?m)`, то:

- символ `^` соответствует началу всей строки или строки текста (сразу после `\n`);
- символ `$` соответствует концу всей строки или строки текста (непосредственно перед `\n`).

С использованием символа $ в многострочном (Multiline) режиме связана одна загвоздка: новая строка в Windows почти всегда обозначается с помощью комбинации \r\n, а не просто \n. Это значит, что в случае $ обычно придется искать совпадение также и с \r, применяя *положительный просмотр вперед*:

```
(?=\r?$)
```

*Положительный просмотр вперед* гарантирует, что \r не станет частью результата. Показанный ниже код соответствует строкам, которые заканчиваются на ".txt":

```
string fileNames = "a.txt" + "\r\n" + "b.doc" + "\r\n" + "c.txt";
string r = @".+\.txt(?=\r?$)";
foreach (Match m in Regex.Matches (fileNames, r, RegexOptions.Multiline))
  Console.Write (m + " ");
```

*ВЫВОД: a.txt c.txt*

Следующий код соответствует всем пустым строкам текста внутри строки s:

```
MatchCollection emptyLines = Regex.Matches (s, "^(?=\r?$)",
                                            RegexOptions.Multiline);
```

Показанный далее код соответствует всем строкам текста, которые либо пусты, либо содержат только пробельные символы:

```
MatchCollection blankLines = Regex.Matches (s, "^[ \t]*(?=\r?$)",
                                            RegexOptions.Multiline);
```

Поскольку привязка соответствует позиции, а не символу, указание одной лишь привязки соответствует пустой строке:

```
Console.WriteLine (Regex.Match ("x", "$").Length);  // 0
```

## Границы слов

Утверждение границы слова \b дает совпадение, когда символы, допустимые в словах (\w), соседствуют с:

- символами, не допустимыми в словах (\W);
- началом/концом строки (^ и $).

\b часто используется для соответствия целым словам. Например:

```
foreach (Match m in Regex.Matches ("Wedding in Sarajevo", @"\b\w+\b"))
  Console.WriteLine (m);
```

*Wedding*
*in*
*Sarajevo*

Следующие операторы подчеркивают эффект от границы слова:

```
int one = Regex.Matches ("Wedding in Sarajevo", @"\bin\b").Count;   // 1
int two = Regex.Matches ("Wedding in Sarajevo", @"in").Count;       // 2
```

В приведенном далее выражении применяется *положительный просмотр вперед* для возвращения слов, за которыми следует символы (sic):

```
string text = "Don't loose (sic) your cool";
Console.Write (Regex.Match (text, @"\b\w+\b\s(?=\(sic\))"));       // loose
```

# Группы

Временами регулярное выражение удобно разделять на последовательности подвыражений, или *группы*. Например, рассмотрим следующее регулярное выражение, которое представляет телефонные номера в США, такие как 206-465-1918:

```
\d{3}-\d{3}-\d{4}
```

Предположим, что мы хотим разделить его на две группы: код зоны и локальный номер. Задачу можно решить, используя круглые скобки для *захвата* каждой группы:

```
(\d{3})-(\d{3}-\d{4})
```

Затем группы можно извлекать программно:

```
Match m = Regex.Match ("206-465-1918", @"(\d{3})-(\d{3}-\d{4})");
Console.WriteLine (m.Groups[1]);  // 206
Console.WriteLine (m.Groups[2]);  // 465-1918
```

Нулевая группа представляет полное совпадение. Другими словами, она имеет то же самое значение, что и свойство `Value` совпадения:

```
Console.WriteLine (m.Groups[0]);  // 206-465-1918
Console.WriteLine (m);            // 206-465-1918
```

Группы являются частью самого языка регулярных выражений. Это означает, что вы можете ссылаться на группу внутри регулярного выражения. Синтаксис `\n` позволяет индексировать группу по ее номеру `n` в рамках выражения. Например, выражение `(\w)ee\1` дает совпадения для `deed` и `peep`. В следующем примере мы ищем в строке все слова, начинающиеся и заканчивающиеся на ту же самую букву:

```
foreach (Match m in Regex.Matches ("pop pope peep", @"\b(\w)\w+\1\b"))
  Console.Write (m + " ");          // pop peep
```

Скобки вокруг `\w` указывают механизму регулярных выражений на необходимость сохранения подсовпадений в группе (одиночной буквы в данном случае), поэтому их можно будет применять позже. В дальнейшем на данную группу можно ссылаться с использованием `\1`, что означает первую группу в выражении.

## Именованные группы

В длинном или сложном выражении работать с группами удобнее по *именам*, а не по индексам. Ниже приведен переписанный предыдущий пример, в котором применяется группа по имени `'letter'`:

```
string regEx =
  @"\b"              +  // граница слова
  @"(?'letter'\w)"   +  // соответствует первой букве; назовем группу 'letter'
  @"\w+"             +  // соответствует промежуточным буквам
  @"\k'letter'"      +  // соответствует последней букве, отмеченной как 'letter'
  @"\b";                // граница слова
foreach (Match m in Regex.Matches ("bob pope peep", regEx))
  Console.Write (m + " ");  // bob peep
```

Вот как назначить имя захваченной группе:

```
(?'имя-группы'выражение-группы)  или  (?<имя-группы>выражение-группы)
```

А вот как ссылаться на группу:

```
\k'имя-группы'  или  \k<имя-группы>
```

В следующем примере производится сопоставление для простого (не вложенного) элемента XML/HTML за счет поиска начального и конечного узлов с совпадающими именами:

```
string regFind =
  @"<(?'tag'\w+?).*>" +   // Соответствует первому дескриптору;
                          //  назовем группу 'tag'
  @"(?'text'.*?)"     +   // Соответствует текстовому содержимому;
                          //   назовем группу 'text'
  @"</\k'tag'>";          // Соответствует последнему дескриптору,
                          //   отмеченному как 'tag'

Match m = Regex.Match ("<h1>hello</h1>", regFind);
Console.WriteLine (m.Groups ["tag"]);   // h1
Console.WriteLine (m.Groups ["text"]);  // hello
```

Анализ всех возможных вариаций в структуре XML, таких как вложенные элементы, является более сложным. Механизм регулярных выражений .NET имеет расширение под названием "соответствующие сбалансированные конструкции", которое может помочь в обработке вложенных дескрипторов — информация о нем доступна в Интернете, а также в книге Джеффри Фридла *Mastering Regular Expressions*.

# Замена и разделение текста

Метод `RegEx.Replace` работает подобно `string.Replace` за исключением того, что использует регулярное выражение.

Следующий код заменяет строку `cat` строкой `dog`. В отличие от `string.Replace` слово `catapult` не будет изменено на `dogapult`, потому что совпадения ищутся по границам слов:

```
string find = @"\bcat\b";
string replace = "dog";
Console.WriteLine (Regex.Replace ("catapult the cat", find, replace));
```

*ВЫВОД: catapult the dog*

Строка замены может ссылаться на исходное совпадение посредством подстановочной конструкции `$0`. В следующем примере числа внутри строки помещаются в угловые скобки:

```
string text = "10 plus 20 makes 30";
Console.WriteLine (Regex.Replace (text, @"\d+", @"<$0>"));
```

*ВЫВОД: <10> plus <20> makes <30>*

Обращаться к захваченным группам можно с помощью конструкций `$1`, `$2`, `$3` и т.д. или `${`*имя*`}` для именованных групп. Чтобы продемонстрировать, когда это может быть удобно, вспомним регулярное выражение из предыдущего раздела, соответствующее простому элементу XML. За счет перестановки групп мы можем сформировать выражение замены, которое перемещает содержимое элемента в атрибут XML:

```
string regFind =
  @"<(?'tag'\w+?).*>" +   // Соответствует первому дескриптору;
                          //   назовем группу 'tag'
  @"(?'text'.*?)"     +   // Соответствует текстовому содержимому;
                          //   назовем группу 'text'
  @"</\k'tag'>";          // Соответствует последнему дескриптору,
                          //   отмеченному как 'tag'
```

```
string regReplace =
  @"<${tag}"          +  // <tag
  @"value="""         +  // value="
  @"${text}"          +  // text
  @"""/>";               // "/>

Console.Write (Regex.Replace ("<msg>hello</msg>", regFind, regReplace));
```

Вот результат:

```
<msg value="hello"/>
```

## Делегат MatchEvaluator

Метод `Replace` имеет перегруженную версию, принимающую делегат `MatchEvaluator`, который вызывается для каждого совпадения. Это позволяет поручить построение содержимого строки замены коду C#, если язык регулярных выражений в такой ситуации оказывается недостаточно выразительным. Например:

```
Console.WriteLine (Regex.Replace ("5 is less than 10", @"\d+",
                   m => (int.Parse (m.Value) * 10).ToString()) );
```

*ВЫВОД: 50 is less than 100*

В рецептурном справочнике мы покажем, как применять `MatchEvaluator` с целью защиты символов Unicode специально для HTML-разметки.

## Разделение текста

Статический метод `Regex.Split` представляет собой более мощную версию метода `string.Split` с регулярным выражением, обозначающим образец разделителя. В следующем примере мы разделяем строку, в которой разделителем считается любая цифра:

```
foreach (string s in Regex.Split ("a5b7c", @"\d"))
  Console.Write (s + " ");  // a b c
```

Результат не содержит сами разделители. Тем не менее, включить разделители можно, поместив выражение внутрь *положительного просмотра вперед*. Следующий код разбивает строку в верблюжьем стиле на отдельные слова:

```
foreach (string s in Regex.Split ("oneTwoThree", @"(?=[A-Z])"))
  Console.Write (s + " ");  // one Two Three
```

# Рецептурный справочник по регулярным выражениям

## Рецепты

### Соответствие номеру карточки социального страхования или телефонному номеру в США

```
string ssNum = @"\d{3}-\d{2}-\d{4}";

Console.WriteLine (Regex.IsMatch ("123-45-6789", ssNum));  // True

string phone = @"(?x)
  ( \d{3}[-\s] | \(\d{3}\)\s? )
    \d{3}[-\s]?
    \d{4}";
```

```
Console.WriteLine (Regex.IsMatch ("123-456-7890",   phone)); // True
Console.WriteLine (Regex.IsMatch ("(123) 456-7890", phone)); // True
```

### Извлечение пар "имя = значение" (по одной в строке текста)

Обратите внимание на использование в начале директивы (?m):

```
string r = @"(?m)^\s*(?'name'\w+)\s*=\s*(?'value'.*)\s*(?=\r?$)";

string text =
  @"id = 3
    secure = true
    timeout = 30";

foreach (Match m in Regex.Matches (text, r))
  Console.WriteLine (m.Groups["name"] + " is " + m.Groups["value"]);

id is 3 secure is true timeout is 30
```

### Проверка сильных паролей

Следующий код проверяет, что пароль состоит минимум из шести символов и включает цифру, символ или знак пунктуации:

```
string r = @"(?x)^(?=.* ( \d | \p{P} | \p{S} )).{6,}";

Console.WriteLine (Regex.IsMatch ("abc12", r));   // False
Console.WriteLine (Regex.IsMatch ("abcdef", r));  // False
Console.WriteLine (Regex.IsMatch ("ab88yz", r));  // True
```

### Строки текста, содержащие, по крайней мере, 80 символов

```
string r = @"(?m)^.{80,}(?=\r?$)";

string fifty = new string ('x', 50);
string eighty = new string ('x', 80);

string text = eighty + "\r\n" + fifty + "\r\n" + eighty;

Console.WriteLine (Regex.Matches (text, r).Count);  // 2
```

### Разбор даты/времени

Показанное ниже выражение поддерживает разнообразные числовые форматы дат и работает независимо от того, где указан год — в начале или конце. Директива (?x) улучшает читабельность, разрешая применение пробельных символов; директива (?i) отключает чувствительность к регистру символов (для необязательного указателя AM/PM). Затем к компонентам совпадения можно обращаться через коллекцию Groups:

```
string r = @"(?x)(?i)
 (\d{1,4}) [./-]
 (\d{1,2}) [./-]
 (\d{1,4}) [\sT]
 (\d+):(\d+):(\d+) \s? (A\.?M\.?|P\.?M\.?)?";

string text = "01/02/2018 5:20:50 PM";

foreach (Group g in Regex.Match (text, r).Groups)
  Console.WriteLine (g.Value + " ");

01/02/2018 5:20:50 PM 01 02 2018 5 20 50 PM
```

(Разумеется, выражение не проверяет корректность даты/времени.)

### Соответствие римским числам

```
string r =
  @"(?i)\bm*"       +
  @"(d?c{0,3}|c[dm])" +
  @"(l?x{0,3}|x[lc])" +
  @"(v?i{0,3}|i[vx])" +
  @"\b";

Console.WriteLine (Regex.IsMatch ("MCMLXXXIV", r));  // True
```

### Удаление повторяющихся слов

Здесь мы захватываем именованную группу dupe:

```
string r = @"(?'dupe'\w+)\W\k'dupe'";

string text = "In the the beginning...";
Console.WriteLine (Regex.Replace (text, r, "${dupe}"));
```

```
In the beginning
```

### Подсчет слов

```
string r = @"\b(\w|[-'])+\b";

string text = "It's all mumbo-jumbo to me";
Console.WriteLine (Regex.Matches (text, r).Count);  // 5
```

### Соответствие GUID

```
string r =
  @"(?i)\b"         +
  @"[0-9a-fA-F]{8}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{12}"  +
  @"\b";

string text = "Its key is {3F2504E0-4F89-11D3-9A0C-0305E82C3301}.";
Console.WriteLine (Regex.Match (text, r).Index);   // 12
```

### Разбор дескриптора XML/HTML

Класс Regex удобен при разборе фрагментов HTML-разметки – особенно, когда документ может быть сформирован некорректно:

```
string r =
  @"<(?'tag'\w+?).*>"  +  // соответствует первому дескриптору;
                          //   назовем группу 'tag'
  @"(?'text'.*?)"      +  // соответствует текстовому содержимому;
                          //   назовем группу 'text'
  @"</\k'tag'>";          // соответствует последнему дескриптору,
                          //   отмеченному как 'tag'

string text = "<h1>hello</h1>";

Match m = Regex.Match (text, r);

Console.WriteLine (m.Groups ["tag"]);   // h1
Console.WriteLine (m.Groups ["text"]);  // hello
```

### Разделение на слова в верблюжьем стиле

Это требует *положительного просмотра вперед*, чтобы включить разделители в верхнем регистре:

```
string r = @"(?=[A-Z])";

foreach (string s in Regex.Split ("oneTwoThree", r))
  Console.Write (s + " ");  // one Two Three
```

### Получение допустимого имени файла

```
string input = "My \"good\" <recipes>.txt";

char[] invalidChars = System.IO.Path.GetInvalidPathChars();
string invalidString = Regex.Escape (new string (invalidChars));

string valid = Regex.Replace (input, "[" + invalidString + "]", "");
Console.WriteLine (valid);

My good recipes.txt
```

### Защита символов Unicode для HTML

```
string htmlFragment = "© 2007";

string result = Regex.Replace (htmlFragment, @"[\u0080-\uFFFF]",
             m => @"&#" + ((int)m.Value[0]).ToString() + ";");

Console.WriteLine (result);  // &#169; 2007
```

### Преобразование символов в строке запроса HTTP

```
string sample = "C%23 rocks";

string result = Regex.Replace (
   sample,
   @"%[0-9a-f][0-9a-f]",
   m => ((char) Convert.ToByte (m.Value.Substring (1), 16)).ToString(),
   RegexOptions.IgnoreCase
);

Console.WriteLine (result);  // C# rocks
```

### Разбор поисковых терминов Google из журнала веб-статистики

Это должно использоваться в сочетании с предыдущим примером преобразования символов в строке запроса:

```
string sample =
  "http://google.com/search?hl=en&q=greedy+quantifiers+regex&btnG=Search";

Match m = Regex.Match (sample, @"(?<=google\..+search\?.*q=).+?(?=(&|$))");

string[] keywords = m.Value.Split (
  new[] { '+' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string keyword in keywords)
  Console.Write (keyword + " ");  // greedy quantifiers regex
```

# Справочник по языку регулярных выражений

В табл. 26.2–26.12 представлена сводка по грамматике и синтаксису регулярных выражений, которые поддерживаются в реализации .NET.

**Таблица 26.2. Управляющие символы**

| Управляющая последовательность | Описание | Шестнадцатеричный эквивалент |
|---|---|---|
| `\a` | Звуковой сигнал | `\u0007` |
| `\b` | Забой | `\u0008` |
| `\t` | Табуляция | `\u0009` |
| `\r` | Возврат каретки | `\u000A` |
| `\v` | Вертикальная табуляция | `\u000B` |
| `\f` | Перевод страницы | `\u000C` |
| `\n` | Новая строка | `\u000D` |
| `\e` | Отмена | `\u001B` |
| `\`*nnn* | ASCII-символ *nnn* в восьмеричной форме (например, `\n052`) | |
| `\x`*nn* | ASCII-символ *nn* в шестнадцатеричной форме (например, `\x3F`) | |
| `\c`*l* | Управляющий ASCII-символ *l* (например, `\cG` для <*Ctrl+G*>) | |
| `\u`*nnnn* | Unicode-символ *nnnn* в шестнадцатеричной форме (например, `\u07DE`) | |
| `\`*символ* | Непреобразуемый символ | |

Специальный случай: внутри регулярного выражения комбинация `\b` означает границу слова за исключением ситуации, когда находится в наборе `[ ]`, где `\b` означает символ забоя.

**Таблица 26.3. Наборы символов**

| Выражение | Описание | Инверсия ("не") |
|---|---|---|
| `[abcdef]` | Соответствие одиночному символу в списке | `[^abcdef]` |
| `[a-f]` | Соответствие одиночному символу в диапазоне | `[^a-f]` |
| `\d` | Соответствие десятичной цифре<br>То же самое, что и `[0-9]` | `\D` |
| `\w` | Соответствие символу, допустимому в словах (по умолчанию варьируется согласно `CultureInfo.CurrentCulture`; например, в английском языке это то же самое, что и `[a-zA-Z_0-9]`) | `\W` |
| `\s` | Соответствие пробельному символу<br>То же самое, что и `[\n\r\t\f\v ]` | `\S` |
| `\p{`*категория*`}` | Соответствие символу в указанной категории (табл. 26.6) | `\P` |
| `.` | (Стандартный режим.) Соответствие любому символу кроме `\n` | `\n` |
| `.` | (Режим `SingleLine`.) Соответствие любому символу | `\n` |

**Таблица 26.4. Категории символов**

| Категория | Описание |
|---|---|
| `\p{L}` | Буквы |
| `\p{Lu}` | Буквы в верхнем регистре |
| `\p{Ll}` | Буквы в нижнем регистре |
| `\p{N}` | Числа |
| `\p{P}` | Знаки пунктуации |
| `\p{M}` | Диакритические знаки |
| `\p{S}` | Символы |
| `\p{Z}` | Разделители |
| `\p{C}` | Управляющие символы |

**Таблица 26.5. Квантификаторы**

| Квантификатор | Описание |
|---|---|
| `*` | Ноль или больше совпадений |
| `+` | Одно или больше совпадений |
| `?` | Ноль или одно совпадение |
| `{`*n*`}` | В точности *n* совпадений |
| `{`*n*`,}` | По меньшей мере, *n* совпадений |
| `{`*n*`,`*m*`}` | Количество совпадений между *n* и *m* |

К любому квантификатору можно применить суффикс `?`, чтобы сделать его *ленивым*, а не *жадным*.

**Таблица 26.6. Подстановки**

| Выражение | Описание |
|---|---|
| `$0` | Подстановка совпадающего текста |
| `$`*номер-группы* | Подстановка индексированного номера группы внутри совпадающего текста |
| `${`*имя-группы*`}` | Подстановка текстового имени группы внутри совпадающего текста |

Подстановки указываются только внутри образца замены.

**Таблица 26.7. Утверждения нулевой ширины**

| Выражение | Описание |
|---|---|
| ^ | Начало строки (или строки текста в *многострочном* режиме) |
| $ | Конец строки (или строки текста в *многострочном* режиме) |
| \A | Начало строки (*многострочный* режим игнорируется) |
| \z | Конец строки (*многострочный* режим игнорируется) |
| \Z | Конец строки текста или всей строки |
| \G | Место начала поиска |
| \b | На границе слова |
| \B | Не на границе слова |
| (?=*expr*) | Продолжать поиск совпадения, только если выражение *expr* дает совпадение справа (*положительный просмотр вперед*) |
| (?!*expr*) | Продолжать поиск совпадения, только если выражение *expr* не дает совпадение справа (*отрицательный просмотр вперед*) |
| (?<=*expr*) | Продолжать поиск совпадения, только если выражение *expr* дает совпадение слева (*положительный просмотр назад*) |
| (?<!*expr*) | Продолжать поиск совпадения, только если выражение *expr* не дает совпадение слева (*отрицательный просмотр назад*) |
| (?>*expr*) | Подвыражение *expr* дает совпадение один раз, и не было возврата назад |

**Таблица 26.8. Конструкции группирования**

| Синтаксис | Описание |
|---|---|
| (*expr*) | Захват давшего совпадение выражения *expr* в индексированную группу |
| (?*номер*) | Захват совпадающей подстроки в группу с указанным номером |
| (?'*имя*') | Захват совпадающей подстроки в группу с указанным именем |
| (?'*имя1*-*имя2*') | Отменить определение *имя2* и сохранить интервал и текущую группу в *имя1*; если *имя2* не определено, то поиск совпадения возвращается назад; *имя1* является необязательным |
| (?:*expr*) | Незахватываемая группа |

**Таблица 26.9. Обратные ссылки**

| Синтаксис | Описание |
|---|---|
| *индекс* | Ссылка на предыдущую захваченную группу по *индексу* |
| \k<*имя*> | Ссылка на предыдущую захваченную группу по *имени* |

**Таблица 26.10. Перестановки**

| Синтаксис | Описание |
|---|---|
| \| | Логическое "ИЛИ" |
| (?(*expr*)*yes*\|*no*) | Соответствует *yes*, если выражение *expr* дает совпадение; в противном случае соответствует *no* (конструкция *no* является необязательной) |
| (?(*name*)*yes*\|*no*) | Соответствует *yes*, если именованная группа *name* имеет совпадение; в противном случае соответствует *no* (конструкция *no* является необязательной) |

**Таблица 26.11. Вспомогательные конструкции**

| Синтаксис | Описание |
|---|---|
| (?#*комментарий*) | Встроенный комментарий |
| #*комментарий* | Комментарий до конца строки (работает только в режиме `IgnorePatternWhitespace`) |

**Таблица 26.12. Параметры регулярных выражений**

| Параметр | Описание |
|---|---|
| (?i) | Соответствие, нечувствительное к регистру символов (регистр символов "игнорируется") |
| (?m) | Многострочный режим; изменяет ^ и $ так, что они соответствуют началу и концу любой строки текста |
| (?n) | Захватывает только явно именованные или пронумерованные группы |
| (?c) | Компилирует в IL |
| (?s) | Однострочный режим; изменяет значение точки (.) так, что она соответствует любому символу |
| (?x) | Устраняет из образца ненужные пробельные символы |
| (?r) | Поиск справа налево; не может быть указан посреди операции |

# 27

# Компилятор Roslyn

Начиная с версии C# 6, компилятор написан полностью на языке C# и имеет модульную архитектуру, поэтому помимо компиляции исходного кода в исполняемый файл или библиотеку его функциональность можно утилизировать многими другими путями. Известный под названием Roslyn, модульный компилятор облегчает написание инструментов статического анализа и рефакторинга кода, редакторов с подсветкой синтаксиса и автозавершением кода, а также подключаемых модулей для Visual Studio, воспринимающих код C#.

Загрузить библиотеки Roslyn можно с помощью диспетчера NuGet, и предусмотрены пакеты как для C#, так и для VB. Поскольку оба языка разделяют определенную архитектуру, существуют общие зависимости. Идентификатором пакета NuGet для библиотек компилятора C# является `Microsoft.CodeAnalysis.CSharp`.

Исходный код для Roslyn находится в открытом доступе и регламентируется лицензией на открытый исходный код Apache 2 (Apache 2 Open Source License), что предоставляет дополнительные возможности, включая трансформацию C# в специальный или специфический для предметной области язык. Исходный код можно загрузить из веб-сайта GitHub по адресу `https://github.com/dotnet/roslyn`.

На веб-сайте GitHub также размещена документация, примеры и пошаговые демонстрации анализа кода и рефакторинга.

Инфраструктура .NET Framework не поставляется вместе со сборками Roslyn и ее версия `csc.exe` вызывает старый компилятор C# 5. Установка Visual Studio 2017 отображает `csc.exe` на компилятор C# 7 (Roslyn).

В отсутствие Visual Studio 2017 вы все равно можете *программно* обращаться к новому компилятору (и его службам), если загрузите и добавите ссылки на сборки Roslyn. Но инструмент `csc.exe`, поставляемый в составе .NET Framework, будет по-прежнему указывать на компилятор C# 5, пока вы не установите версию Visual Studio 2017.

Ниже перечислены сборки, которые входят в состав библиотеки компилятора C#:

```
Microsoft.CodeAnalysis.dll
Microsoft.CodeAnalysis.CSharp.dll
System.Collections.Immutable.dll
System.Reflection.Metadata.dll
```

Первая сборка также применяется компилятором VB и содержит общие базовые типы для деревьев, символов, объектов компиляции и т.д.

Все листинги кода, приведенные в настоящей главе, доступны в виде интерактивных примеров для утилиты LINQPad. Перейдите на вкладку Samples (Примеры) в окне LINQPad, щелкните на ссылке Download/import more samples (Загрузить/импортировать дополнительные примеры) и в открывшемся окне щелкните на ссылке Download C# 7.0 in a Nutshell samples (Загрузить примеры для книги *C# 7.0 in a Nutshell*).

# Архитектура Roslyn

Архитектура Roslyn разделяет компиляцию на три фазы.

1. Разбор кода в синтаксические деревья (*синтаксический* уровень).
2. Привязка идентификаторов к символам (*семантический* уровень).
3. Выпуск кода IL.

На первой фазе *анализатор* читает код C# и производит *синтаксические деревья*. Синтаксическое дерево – это объектная модель документа (Document Object Model – DOM), которая описывает исходный код в древовидной структуре.

На второй фазе происходит *статическое связывание* C#. Компилятор читает ссылки на сборки и выясняет, например, что "Console" относится к `System.Console` в `mscorlib.dll`. Частью такого процесса также являются распознавание перегруженных версий и выведение типов.

Третья фаза производит выходную сборку. Если вы планируете использовать Roslyn для анализа или рефакторинга кода, то не будете иметь дело с функциональностью третьей фазы.

Редактор Visual Studio применяет выходные данные синтаксического уровня для выделения цветом ключевых слов, строк, комментариев и запрещенного кода (соответственно, синим, красным, зеленым и серым цветами), а выходные данные семантического уровня – для выделения цветом распознанных имен типов (бирюзовым цветом).

## Рабочие области

В этой главе мы описываем компилятор и открываемые им функциональные средства. Полезно помнить о наличии дополнительного "уровня" над компилятором, который называется *рабочими областями*. Он также доступен через диспетчер NuGet; идентификатор пакета выглядит как `Microsoft.CodeAnalysis.CSharp.Workspaces`.

Уровень рабочих областей воспринимает решения, проекты и документы Visual Studio, а также включает дополнительные службы, не относящиеся строго к процессам компиляции, такие как рефакторинг кода.

Уровень рабочих областей поставляется с открытым кодом, исследование которого способствует лучшему пониманию того, что происходит на уровне компиляции.

# Синтаксические деревья

Синтаксическое дерево – это DOM-модель для исходного кода. Следует отметить, что API-интерфейс синтаксических деревьев полностью отделен от API-интерфейса

System.Linq.Expressions, обсуждаемого в разделе "Деревья выражений" главы 8, хотя концептуальные сходства имеются. Оба API-интерфейса могут представлять выражения C# в DOM-модели; однако синтаксическое дерево Roslyn обладает перечисленными ниже уникальными характеристиками.

- Оно способно представлять все конструкции языка C#, а не только выражения.
- Оно может включать комментарии, пробельные символы и другую дополнительную синтаксическую информацию, а также достоверно переключаться обратно на первоначальный исходный код.
- Оно поступает с методом ParseText, который разбирает исходный код в синтаксическое дерево.

С другой стороны, API-интерфейс System.Linq.Expressions обладает следующими уникальными характеристиками.

- Он встроен в .NET Framework, и компилятор C# сам запрограммирован на выпуск типов System.Linq.Expression в случае обнаружения лямбда-выражения с преобразованием в тип Expression<T> при присваивании.
- Он имеет быстрый и легковесный метод Compile, который выпускает делегат. По контрасту семантический уровень, компилирующий синтаксические деревья Roslyn, предлагает только тяжеловесный вариант компиляции полной программы в сборку.

Общей чертой обоих API-интерфейсов является неизменяемость синтаксических деревьев, так что ни один из их элементов не может быть модифицирован после создания. Это значит, что приложения вроде Visual Studio и LINQPad при каждом нажатии вами клавиши в редакторе должны создавать новое синтаксическое дерево для обновления служб подсветки синтаксиса и автозавершения кода. Данный процесс не настолько дорогостоящий, как может показаться, т.к. новое синтаксическое дерево в состоянии повторно использовать большинство элементов из старого синтаксического дерева (как показано в разделе "Трансформация синтаксического дерева" далее в главе). К тому же знание того, что объект не может изменяться, упрощает работу с API-интерфейсом. Неизменяемость также делает возможным более легкое и быстрое распараллеливание, поскольку многопоточный код может безопасно получать доступ ко всем частям синтаксического дерева без блокировок.

## Структура SyntaxTree

Структура SyntaxTree содержит три главных элемента.

- *Узлы* (Абстрактный класс SyntaxNode). Представляет такие конструкции C#, как выражения, операторы и объявления методов. Узлы всегда имеют, по крайней мере, один дочерний элемент, а потому узел никогда не может быть листовым в дереве. В качестве дочерних элементов узлы могут иметь узлы и лексемы.
- *Лексемы* (Структура SyntaxToken). Представляет идентификаторы, ключевые слова, операции и знаки пунктуации, которые являются частью исходного кода. Единственный вид дочерних элементов, которые могут иметь лексемы – это ведущая и замыкающая дополнительная синтаксическая информация. Родительским элементом лексемы всегда будет узел.
- *Дополнительная синтаксическая информация* (Структура SyntaxTrivia). Под дополнительной синтаксической информацией понимаются пробельные символы,

комментарии, директивы препроцессора и код, который остается неактивным вследствие условной компиляции. Дополнительная синтаксическая информация всегда ассоциирована с лексемой, которая находится непосредственно слева или справа, и доступна через свойства `TrailingTrivia` и `LeadingTrivia` данной лексемы соответственно.

На рис. 27.1 показана структура следующего кода, где узлы изображаются черным, лексемы – серым, а дополнительная синтаксическая информация – белым цветом:

```
Console.WriteLine ("Hello");
```

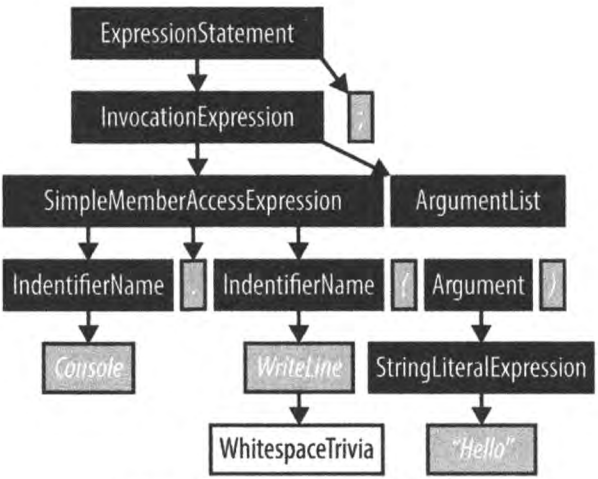



***Рис. 27.1.*** *Синтаксическое дерево*

Класс `SyntaxNode` является абстрактным, а для каждого синтаксического элемента в языке C# предусмотрен его специальный подкласс, такой как `VariableDeclarationSyntax` или `TryStatementSyntax`.

Поскольку `SyntaxToken` и `SyntaxTrivia` – структуры, то все разновидности лексем и дополнительной синтаксической информации представляются с помощью единственного типа. Для различения видов лексем или дополнительной синтаксической информации должно применяться свойство `RawKind` или расширяющий метод `Kind` (как объясняется в следующем разделе).

Лучший способ исследования синтаксического дерева предполагает использование визуализатора. В среде Visual Studio имеется загружаемый визуализатор для применения с ее отладчиком, а LINQPad располагает встроенным визуализатором. Инструмент LINQPad автоматически отображает визуализатор для кода в текстовом редакторе, когда вы щелкаете на кнопке **Tree** (Дерево) в окне вывода. Кроме того, вы можете запросить у LINQPad отображение визуализатора для синтаксического дерева, созданного вами программно, путем вызова метода `DumpSyntaxTree` на объекте дерева (или метода `DumpSyntaxNode` на объекте узла).

### Понятие типов узлов

Для отражения результата синтаксического разбора были спроектированы подклассы класса SyntaxNode, которые не замечают семантическую информацию о типе/символе, получаемую на более позднем этапе связывания. Например, рассмотрим результат разбора следующего кода:

```
using System;

class Foo : SomeBaseClass
{
  void Test() { Console.WriteLine(); }
}
```

Можно было бы ожидать, что вызов Console.WriteLine должен быть представлен классом по имени MethodCallExpressionSyntax, но такого класса не существует. Взамен вызов представляется с помощью объекта InvocationExpressionSyntax, под которым есть объект SimpleMemberAccessExpression. Причина в том, что анализатор не осведомлен о типах, поэтому он не знает, что Console является типом, а WriteLine — методом. Есть много других возможностей: Console могло бы быть свойством класса SomeBaseClass или WriteLine могло бы быть событием, полем либо свойством какого-то типа делегата. Из синтаксиса можно узнать лишь то, что выполняется доступ к члену (*идентификатор.идентификатор*), за которым следует разновидность *вызова* с нулевым количеством аргументов.

## Общие свойства и методы

Узлы, лексемы и дополнительная синтаксическая информация имеют несколько важных общих свойств и методов.

- *Свойство* `SyntaxTree`. Возвращает синтаксическое дерево, к которому принадлежит объект.
- *Свойство* `Span`. Возвращает позицию объекта в исходном коде (см. раздел "Нахождение дочернего элемента по его смещению" далее в главе).
- *Расширяющий метод* `Kind`. Возвращает член перечисления SyntaxKind, классифицирующего узел, лексему или дополнительную синтаксическую информацию посредством нескольких сотен значений (например, IntKeyword, CommaToken и WhitespaceTrivia). Одно и то же перечисление SyntaxKind охватывает узлы, лексемы или дополнительную синтаксическую информацию.
- *Метод* `ToString`. Возвращает текст (исходный код) для узла, лексемы или дополнительной синтаксической информации. В случае лексем его эквивалентом является свойство Text.
- *Метод* `GetDiagnostics`. Возвращает ошибки или предупреждения, сгенерированные во время разбора.
- *Метод* `IsEquivalentTo`. Возвращает true, если объект идентичен другому экземпляру узла, лексемы или дополнительной синтаксической информации. Отличия в пробельных символах существенны (чтобы проигнорировать пробельные символы, перед сравнением вызовите метод NormalizeWhitespace).

Узлы и лексемы также располагают свойством FullSpan и методом ToFullString. Они принимают во внимание дополнительную синтаксическую информацию, тогда как Span и ToString — нет.

Расширяющий метод `Kind` — это сокращение для приведения свойства `RawKind`, имеющего тип `int`, к типу `Microsoft.CodeAnalysis.CSharp.SyntaxKind`. Причина, по которой не было просто определено свойство `Kind` типа `SyntaxKind`, связана с тем, что типы лексем и дополнительной синтаксической информации также используются в синтаксических деревьях VB, имеющих другой тип перечисления для `SyntaxKind`.

## Получение синтаксического дерева

Статический метод `ParseText` класса `CSharpSyntaxTree` производит разбор кода C# в объект `SyntaxTree`:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText (@"class Test
{
  static void Main() => Console.WriteLine (""Hello"");
}");

Console.WriteLine (tree.ToString());

tree.DumpSyntaxTree();  // Отображает визуализатор синтаксических деревьев
                        // в LINQPad
```

Чтобы выполнить такой код в проекте Visual Studio, установите NuGet-пакет `Microsoft.CodeAnalysis.CSharp` и импортируйте следующие пространства имен:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
```

Методу `ParseText` можно дополнительно передавать объект `CSharpParseOptions`, задающий версию языка C#, символы препроцессора и член перечисления `DocumentationMode`, который указывает на то, должны ли быть подвергнуты разбору XML-комментарии (см. раздел "Структурированная дополнительная синтаксическая информация" далее в главе). Также есть возможность указать значение перечисления `SourceCodeKind`. Выбор значения `Script` заставляет анализатор вместо требования полной программы принимать одиночное выражение или оператор (операторы), что поддерживается в Roslyn 2 и последующих версиях.

Еще один способ получения синтаксического дерева предусматривает вызов метода `CSharpSyntaxTree.Create` с передачей ему объектного графа узлов и лексем. Мы покажем, как создавать такие объекты, в разделе "Трансформация синтаксического дерева" далее в главе.

После разбора дерева можно получить ошибки и предупреждения, вызвав метод `GetDiagnostics`. (Упомянутый метод можно также вызывать для отдельного узла или лексемы.)

Если разбор привел к непредвиденным ошибкам, тогда структура дерева может оказаться не той, которая ожидалась. По этой причине полезно вызывать метод `GetDiagnostics`, прежде чем двигаться дальше.

Полезная особенность дерева с ошибками заключается в том, что оно будет переключаться обратно на первоначальный текст (с теми же самыми ошибками). В таких случаях анализатор делает все возможное, чтобы предоставить синтаксическое дерево, которое пригодно для семантического уровня, при необходимости создавая "фантомные узлы". Такой подход позволяет инструментам вроде автозавершения кода работать с неполным кодом. (Выяснить, является ли узел фантомным, можно с помощью свойства `IsMissing`.)

Вызов метода GetDiagnostics на синтаксическом дереве, созданном в предыдущем разделе, указывает на отсутствие ошибок, несмотря на вызов Console.WriteLine без импортирования пространства имен System. Это хороший пример сравнения синтаксического и семантического разборов: программа синтаксически корректна и ошибка не проявится до тех пор, пока мы не соберем все вместе, добавим ссылки на сборки и запросим *семантическую модель*, где происходит связывание.

## Обход и поиск в дереве

Структура SyntaxTree действует в качестве оболочки для древовидной структуры. Она имеет ссылку на единственный корневой узел, который можно получить с помощью вызова метода GetRoot:

```
var tree = CSharpSyntaxTree.ParseText (@"class Test
{
  static void Main() => Console.WriteLine (""Hello"");
}");
SyntaxNode root = tree.GetRoot();
```

Корневой узел программы C# представлен объектом CompilationUnitSyntax:

```
Console.WriteLine (root.GetType().Name);  // CompilationUnitSyntax
```

### Обход дочерних элементов

Класс SyntaxNode предлагает дружественные к LINQ методы для обхода своих дочерних узлов и лексем. Вот простейшие из них:

```
IEnumerable<SyntaxNode> ChildNodes()
IEnumerable<SyntaxToken> ChildTokens()
```

Из предыдущего примера следует, что наш корневой узел имеет единственный дочерний узел типа ClassDeclarationSyntax:

```
var cds = (ClassDeclarationSyntax) root.ChildNodes().Single();
```

Мы можем выполнить перечисление членов объекта cds либо через его метод ChildNodes, либо посредством свойства Members класса ClassDeclarationSyntax:

```
foreach (MemberDeclarationSyntax member in cds.Members)
  Console.WriteLine (member.ToString());
```

с показанным ниже результатом:

```
static void Main() => Console.WriteLine (""Hello"");
```

Существуют также методы Descendant*, которые рекурсивно спускаются к дочерним элементам. Реализовать перечисление лексем, составляющих нашу программу, можно следующим образом:

```
foreach (var token in root.DescendantTokens())
  Console.WriteLine ($"{token.Kind(),-30} {token.Text}");
```

Вот результат:

```
ClassKeyword                   class
IdentifierToken                Test
OpenBraceToken                 {
StaticKeyword                  static
VoidKeyword                    void
IdentifierToken                Main
OpenParenToken                 (
```

```
CloseParenToken              )
EqualsGreaterThanToken       =>
IdentifierToken              Console
DotToken                     .
IdentifierToken              WriteLine
OpenParenToken               (
StringLiteralToken           "Hello"
CloseParenToken              )
SemicolonToken               ;
CloseBraceToken              }
EndOfFileToken
```

Обратите внимание на отсутствие пробельных символов в результате. Замена обращения `token.Text` вызовом метода `token.ToFullString` привела бы к получению пробельных символов (и любой другой дополнительной синтаксической информации).

В приведенном ниже коде метод `DescendantNodes` используется при нахождении местоположения узла для нашего объявления метода:

```
var ourMethod = root.DescendantNodes()
                .First (m => m.Kind() == SyntaxKind.MethodDeclaration);
```

Или по-другому:

```
var ourMethod = root.DescendantNodes()
                .OfType<MethodDeclarationSyntax>()
                .Single();
```

В последнем примере переменная `ourMethod` имеет тип `MethodDeclarationSyntax`, который открывает доступ к полезным свойствам, специфичным для объявлений методов. Скажем, если бы в примере содержалось более одного определения метода, и нужно было найти только метод с именем `Main`, то можно было бы поступить так:

```
var mainMethod = root.DescendantNodes()
                 .OfType<MethodDeclarationSyntax>()
                 .Single (m => m.Identifier.Text == "Main");
```

`Identifier` — это свойство класса `MethodDeclarationSyntax`, которое возвращает лексему, соответствующую идентификатору метода (т.е. его имени). Получить тот же самый результат можно и с большими усилиями:

```
root.DescendantNodes().First (m =>
  m.Kind() == SyntaxKind.MethodDeclaration &&
  m.ChildTokens().Any (t =>
    t.Kind() == SyntaxKind.IdentifierToken && t.Text == "Main"));
```

В классе `SyntaxNode` также определены методы `GetFirstToken` и `GetLastToken`, которые являются эквивалентами вызова методов `DescendantTokens().First` и `DescendantTokens().Last`.

Метод `GetLastToken` быстрее `DescendantTokens().Last`, т.к. он возвращает прямую ссылку, а не производит перечисление по всем потомкам.

Поскольку узлы могут содержать как дочерние узлы, так и лексемы, относительный порядок следования которых имеет значение, то существуют также методы для их перечисления вместе:

```
ChildSyntaxList ChildNodesAndTokens()
IEnumerable<SyntaxNodeOrToken> DescendantNodesAndTokens()
IEnumerable<SyntaxNodeOrToken> DescendantNodesAndTokensAndSelf()
```

(Класс ChildSyntaxList реализует интерфейс IEnumerable<SyntaxNodeOrToken>, одновременно открывая доступ к свойству Count и индексатору для обращения к элементам по позициям.)

Обходить дополнительную синтаксическую информацию можно напрямую из узла с помощью методов GetLeadingTrivia, GetTrailingTrivia и DescendantTrivia. Однако чаще всего вы будете производить доступ к дополнительной синтаксической информации через лексему, к которой она присоединена, посредством свойств LeadingTrivia и TrailingTrivia лексемы. Или же при преобразовании в текст вы могли бы применять метод ToFullString, который включает дополнительную синтаксическую информацию в результат.

### Обход родительских элементов

Узлы и лексемы имеют свойство Parent типа SyntaxNode.

Для структуры SyntaxTrivia "родительским элементом" является лексема, доступная через свойство Token.

Узлы также располагают методами, которые поднимаются вверх по дереву; их имена снабжены префиксом Ancestor.

### Нахождение дочернего элемента по его смещению

Все узлы, лексемы и дополнительная синтаксическая информация имеют свойство Span типа TextSpan для указания смещений начала и конца в исходном коде. В узлах и лексемах также есть свойство FullSpan, которое включает ведущую и замыкающую дополнительную синтаксическую информацию (тогда как свойство Span ее не содержит). Тем не менее, свойство Span узла включает дочерние узлы и лексемы.

---

**Работа со структурой TextSpan**

Структура TextSpan имеет целочисленные свойства Start, Length и End, которые указывают символьные смещения в исходном коде. В ней также определены методы, такие как Overlap, OverlapsWith, Intersection и IntersectsWith. Разница между перекрытием и пересечением сводится к одному символу: два промежутка *перекрываются*, если один начинается до окончания другого (<), тогда как они *пересекаются*, когда просто соприкасаются (<=).

Класс SyntaxTree открывает доступ к методу GetLineSpan, который преобразует структуру TextSpan в строковое и символьное смещение. Метод GetLineSpan игнорирует результаты действия любых директив #line, присутствующих в исходном коде. Есть также метод GetMappedLineSpan, который принимает во внимание упомянутые директивы.

---

С помощью методов FindNode, FindToken и FindTrivia класса SyntaxNode можно искать объект-потомок по позиции. Эти методы возвращают объект-потомок с наименьшим промежутком, который полностью содержит промежуток, указанный при вызове. Существует также метод ChildThatContainsPosition, производящий поиск и узлов-потомков, и лексем.

Если поиск дает в результате два узла с идентичными промежутками (обычно дочерний и внучатый узлы), то метод FindNode возвратит внешний (родительский) узел. Такое поведение можно изменить, передав методу getInnermostNodeForTie значение true в дополнительном аргументе.

Методы Find* также принимают необязательный параметр findInsideTrivia типа bool. В случае передачи true методы выполняют поиск узлов или лексем также и внутри *структурированной дополнительной синтаксической информации* (см. раздел "Дополнительная синтаксическая информация"далее в главе).

### CSharpSyntaxWalker

Еще один способ обхода дерева предполагает создание подкласса класса CSharpSyntaxWalker с переопределением одного или более из его сотен виртуальных методов. Следующий класс подсчитывает количество операторов if:

```
class IfCounter : CSharpSyntaxWalker
{
  public int IfCount { get; private set; }

  public override void VisitIfStatement (IfStatementSyntax node)
  {
    IfCount++;
    // Вызвать базовый метод, если нужно спуститься к дочерним элементам.
    base.VisitIfStatement (node);
  }
}
```

Вот как его задействовать:

```
var ifCounter = new IfCounter ();
ifCounter.Visit (root);
Console.WriteLine ($"I found {ifCounter.IfCount} if statements");
```

Результат эквивалентен выполнению такого кода:

```
root.DescendantNodes().OfType<IfStatementSyntax>().Count()
```

Написание средства обхода синтаксиса может оказаться легче, чем использование методов Descendant* в более сложных случаях, когда необходимо переопределять множество методов (отчасти потому, что C# не обладает возможностью сопоставления по образцу, присущей языку F#).

По умолчанию класс CSharpSyntaxWalker посещает только узлы. Чтобы посетить лексемы или дополнительную синтаксическую информацию, потребуется вызвать базовый конструктор со значением перечисления SyntaxWalkerDepth, указывающим желаемую глубину (узел лексема дополнительная синтаксическая информация). Затем можно переопределять методы VisitToken и VisitTrivia:

```
class WhiteWalker : CSharpSyntaxWalker  // Посчитывает пробельные символы
{
  public int SpaceCount { get; private set; }

  public WhiteWalker() : base (SyntaxWalkerDepth.Trivia) { }

  public override void VisitTrivia (SyntaxTrivia trivia)
  {
    SpaceCount += trivia.ToString().Count (char.IsWhiteSpace);
    base.VisitTrivia (trivia);
  }
}
```

Если удалить вызов базового конструктора из конструктора WhiteWalker, то метод VisitTrivia не запустится.

### Дополнительная синтаксическая информация

Дополнительная синтаксическая информация предназначена для кода, который после разбора компилятор может почти полностью игнорировать в смысле построения выходной сборки. Сюда входят пробельные символы, комментарии, XML-документация, директивы препроцессора и код, являющийся неактивным из-за условной компиляции.

Обязательные пробельные символы в коде также рассматриваются как дополнительная синтаксическая информация. Хотя она жизненно важна для разбора, после производства синтаксического дерева необходимость в ней отпадает (во всяком случае, со стороны компилятора). Дополнительная синтаксическая информация по-прежнему важна для переключения обратно на первоначальный исходный код.

Дополнительная синтаксическая информация принадлежит лексеме, с которой она соседствует. По соглашению анализатор помещает пробельные символы и комментарии, следующие за лексемой, в конец строки (внутрь замыкающей дополнительной синтаксической информации лексемы). Все, что находится после этого, анализатор трактует как ведущую дополнительную синтаксическую информацию для следующей лексемы. (Существуют исключения для позиций в самом начале и в конце файла.) Если вы создаете лексемы программно (см. раздел "Трансформация синтаксического дерева" далее в главе), то можете помещать пробельные символы в любое место (или вообще не поступать так, если не собираетесь выполнять преобразование обратно в исходный код):

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static /*комментарий*/ void Main() {}
}");

SyntaxNode root = tree.GetRoot();

// Найти лексему ключевого слова static:
var method = root.DescendantTokens().Single (t =>
  t.Kind() == SyntaxKind.StaticKeyword);

// Вывести дополнительную синтаксическую информацию вокруг лексемы
// ключевого слова static:
foreach (SyntaxTrivia t in method.LeadingTrivia)
  Console.WriteLine (new { Kind = "Leading " + t.Kind(), t.Span.Length });

foreach (SyntaxTrivia t in method.TrailingTrivia)
  Console.WriteLine (new { Kind = "Trailing " + t.Kind(), t.Span.Length });
```

Ниже показан вывод:

```
{ Kind = Leading WhitespaceTrivia, Length = 1 }
{ Kind = Trailing WhitespaceTrivia, Length = 1 }
{ Kind = Trailing MultiLineCommentTrivia, Length = 11 }
{ Kind = Trailing WhitespaceTrivia, Length = 1 }
```

### Директивы препроцессора

Может показаться странным, что директивы препроцессора считаются дополнительной синтаксической информацией, учитывая значительное воздействие на вывод, которое оказывают некоторые директивы (в частности, директивы условной компиляции).

Причина в том, что директивы препроцессора семантически обрабатываются самим анализатором, т.е. выполнение предварительной обработки — задача анализато-

ра. После этого не остается ничего такого, что компилятор должен явно принимать во внимание (кроме #pragma). В целях иллюстрации давайте посмотрим, как анализатор обрабатывает директивы условной компиляции:

```
#define FOO

#if FOO
   Console.WriteLine ("FOO is defined");
#else
   Console.WriteLine ("FOO is not defined");
#endif
```

Прочитав директиву #if FOO, анализатор знает, что символ FOO определен, поэтому следующая за ней строка разбирается обычным образом (как узлы и лексемы), в то время как строка кода, находящаяся за директивой #else, разбирается в DisabledTextTrivia.

При вызове метода CSharpSyntaxTree.Parse можно предоставить дополнительные символы препроцессора, сконструировав и передав данному методу экземпляр CSharpParseOptions.

Следовательно, благодаря условной компиляции это именно тот текст, который может быть проигнорирован и находится в конце дополнительной синтаксической информации (т.е. неактивный код и сами директивы условной компиляции).

Директива #line обрабатывается аналогично в том смысле, что она читается и интерпретируется анализатором. Собранная информация применяется при вызове метода GetMappedLineSpan на синтаксическом дереве.

Директива #region семантически пуста: единственная роль анализатора заключается в проверке того, что директивам #region соответствуют директивы #endregion. Директивы #error и #warning также обрабатываются анализатором и приводят к генерации ошибок и предупреждений, которые можно просмотреть, вызвав метод GetDiagnostics на дереве или узле.

Исследование содержимого директив препроцессора может быть в равной степени полезным и для целей, выходящих за рамки производства выходной сборки (скажем, подсветка синтаксиса). Это облегчается посредством *структурированной дополнительной синтаксической информации.*

### Структурированная дополнительная синтаксическая информация

Существуют два вида дополнительной синтаксической информации.

- *Неструктурированная дополнительная синтаксическая информация.* Комментарии, пробельные символы и код, который неактивен из-за условной компиляции.
- *Структурированная дополнительная синтаксическая информация.* Директивы препроцессора и XML-документация.

Неструктурированная дополнительная синтаксическая информация трактуется исключительно как текст. С другой стороны, структурированная дополнительная синтаксическая информация располагает также и содержимым, которое разбирается в миниатюрное синтаксическое дерево.

Свойство HasStructure структуры SyntaxTrivia указывает, присутствует ли структурированная дополнительная синтаксическая информация, а метод GetStructure возвращает корневой узел для миниатюрного синтаксического дерева:

```
var tree = CSharpSyntaxTree.ParseText (@"#define FOO");

// В LINQPad:
tree.DumpSyntaxTree();  // LINQPad отображает структурированную дополнительную
                        // синтаксическую информацию в визуализаторе.

SyntaxNode root = tree.GetRoot();

var trivia = root.DescendantTrivia().First();
Console.WriteLine (trivia.HasStructure);           // True
Console.WriteLine (trivia.GetStructure().Kind());  // DefineDirectiveTrivia
```

В случае директив препроцессора можно переходить непосредственно к структурированной дополнительной синтаксической информации, вызывая метод `GetFirstDirective` класса `SyntaxNode`. Имеется также свойство `ContainsDirectives`, предназначенное для указания на наличие синтаксической информации препроцессора:

```
var tree = CSharpSyntaxTree.ParseText (@"#define FOO");
SyntaxNode root = tree.GetRoot();

Console.WriteLine (root.ContainsDirectives);    // True

// directive - это корневой узел структурированной дополнительной
// синтаксической информации:
var directive = root.GetFirstDirective();
Console.WriteLine (directive.Kind());           // DefineDirectiveTrivia
Console.WriteLine (directive.ToString());       // #define FOO

// Если директив было больше, тогда мы можем получить их следующим образом:
Console.WriteLine (directive.GetNextDirective());   // (null)
```

После получения узла дополнительной синтаксической информации мы можем привести его к специфичному типу и обращаться к свойствам, как поступали бы с любым другим узлом:

```
var hashDefine = (DefineDirectiveTriviaSyntax) root.GetFirstDirective();
Console.WriteLine (hashDefine.Name.Text);           // FOO
```

Все узлы, лексемы и дополнительная синтаксическая информация имеют свойство `IsPartOfStructuredTrivia`, которое указывает на то, является ли данный объект частью дерева структурированной дополнительной синтаксической информации (т.е. происходит от объекта дополнительной синтаксической информации).

## Трансформация синтаксического дерева

С помощью набора методов (большинство из которых представляют собой расширяющие методы) со следующими префиксами можно "модифицировать" узлы, лексемы и дополнительную синтаксическую информацию:

```
Add*
Insert*
Remove*
Replace*
With*
Without*
```

Поскольку синтаксические деревья являются неизменяемыми, все эти методы возвращают новый объект с желаемыми модификациями, оставляя исходный объект незатронутым.

### Обработка изменений в исходном коде

Если вы строите, скажем, редактор кода C#, то должны обновлять синтаксическое дерево на основе изменений, внесенных в исходный код. Класс `SyntaxTree` имеет метод `WithChangedText`, выполняющий именно такую работу: он частично разбирает исходный код заново, базируясь на модификациях, которые описаны с помощью экземпляра класса `SourceText` (из пространства имен `Microsoft.CodeAnalysis.Text`).

Чтобы создать экземпляр класса `SourceText`, вызовите его статический метод `From`, передав ему завершенный исходный код. Затем для создания синтаксического дерева можно использовать приведенный ниже код:

```
SourceText sourceText = SourceText.From ("class Program {}");
var tree = CSharpSyntaxTree.ParseText (sourceText);
```

В качестве альтернативы экземпляр класса `SourceText` можно получить для существующего дерева, вызвав его метод `GetText`.

Теперь `sourceText` можно "обновить" с помощью метода `Replace` или `WithChanges`. Например, вот как заменить первые пять символов (`class`) символами `struct`:

```
var newSource = sourceText.Replace (0, 5, "struct");
```

Наконец, можно вызвать метод `WithChangedText` на дереве для его обновления:

```
var newTree = tree.WithChangedText (newSource);
Console.WriteLine (newTree.ToString());  // struct Program {}
```

### Создание новых узлов, лексем и дополнительной синтаксической информации с помощью класса `SyntaxFactory`

Статические методы класса `SyntaxFactory` позволяют программно создавать узлы, лексемы и дополнительную синтаксическую информацию, которую можно применять для "трансформации" существующих синтаксических деревьев или для построения новых деревьев с нуля.

Самой трудной частью этого процесса является выяснение того, узлы и лексемы какого вида нужно создавать. Решение предусматривает предварительный разбор желаемого примера кода с целью исследования результатов в визуализаторе синтаксиса. Для примера представим, что необходимо создать синтаксический узел для следующего кода:

```
using System.Text;
```

Визуализировать синтаксическое дерево для этого кода в LINQPad можно так:

```
CSharpSyntaxTree.ParseText ("using System.Text;").DumpSyntaxTree();
```

(Мы можем провести разбор кода `using System.Text;` без ошибок, потому что он допустим как завершенная, хотя и функционально пустая программа. Большинство других фрагментов кода потребуется помещать внутрь определения метода и/или типа, чтобы их разбор стал возможным.)

Результат имеет показанную ниже структуру, в которой нас интересует второй узел (т.е. `UsingDirective` и его потомки):

```
Kind                              Token Text
================================  ==========
CompilationUnit (node)
  UsingDirective (node)
    UsingKeyword (token)          using
      WhitespaceTrivia (trailing)
```

```
  QualifiedName (node)
    IdentifierName (node)
      IdentifierToken (token)     System
    DotToken (token)              .
    IdentifierName (node)
      IdentifierToken (token)     Text
  SemiColonToken (token)          ;
EndOfFileToken (token)
```

Начав изнутри, мы обнаруживаем два узла `IdentifierName`, родительским узлом которых является `QualifiedName`, что можно создать следующим образом:

```
QualifiedNameSyntax qualifiedName = SyntaxFactory.QualifiedName (
  SyntaxFactory.IdentifierName ("System"),
  SyntaxFactory.IdentifierName ("Text"));
```

Мы используем перегруженную версию метода `QualifiedName`, принимающую два идентификатора. Она вставляет лексему точки автоматически.

Теперь узел необходимо поместить внутрь `UsingDirective`:

```
UsingDirectiveSyntax usingDirective =
  SyntaxFactory.UsingDirective (qualifiedName);
```

Поскольку мы не указываем лексемы для ключевого слова `using` или завершающей точки с запятой, такие лексемы создаются и добавляются автоматически. Тем не менее, автоматически создаваемые лексемы не включают пробельные символы. Это не препятствует компиляции, но преобразование дерева в строку даст в результате синтаксически некорректный код:

```
Console.WriteLine (usingDirective.ToFullString()); // usingSystem.Text;
```

Устранить проблему можно вызовом метода `NormalizeWhitespace` на узле (или одном из его предшественников); такое действие автоматически добавляет дополнительную синтаксическую информацию для пробельных символов (обеспечивая корректность и читабельность). Или же в целях большего контроля пробельные символы можно добавить явно:

```
usingDirective = usingDirective.WithUsingKeyword (
  usingDirective.UsingKeyword.WithTrailingTrivia (
    SyntaxFactory.Whitespace (" ")));

Console.WriteLine (usingDirective.ToFullString()); // using System.Text;
```

Для краткости мы задействуем существующую лексему `UsingKeyword` узла, к которому добавляем замыкающую дополнительную синтаксическую информацию. Мы могли бы создать эквивалентную лексему с бо́льшими усилиями, вызвав `SyntaxFactory.Token(SyntaxKind.UsingKeyword)`.

Финальный шаг заключается в добавлении узла `UsingDirective` к существующему или новому синтаксическому дереву (или, выражаясь точнее, к корневому узлу дерева). Для этого мы приводим корневой узел существующего дерева к типу `CompilationUnitSyntax` и вызываем метод `AddUsings`. Затем мы можем создать новое дерево из трансформированной единицы компиляции:

```
var existingTree = CSharpSyntaxTree.ParseText ("class Program {}");
var existingUnit = (CompilationUnitSyntax) existingTree.GetRoot();

var unitWithUsing = existingUnit.AddUsings (usingDirective);

var treeWithUsing = CSharpSyntaxTree.Create (
  unitWithUsing.NormalizeWhitespace());
```

Помните, что все части синтаксического дерева являются неизменяемыми. Вызов AddUsings возвращает новый узел, оставляя исходный узел незатронутым. Игнорирование возвращаемого значения – путь к ошибке!

Так как мы вызвали метод NormalizeWhitespace на единице компиляции, вызов ToString на дереве выдаст синтаксически корректный и читабельный код. В качестве альтернативы мы могли бы добавить к usingDirective явную дополнительную синтаксическую информацию для новой строки:

```
.WithTrailingTrivia (SyntaxFactory.EndOfLine("\r\n\r\n"))
```

Создание единицы компиляции и синтаксического дерева с нуля представляет собой похожий процесс. Простейший подход состоит в том, чтобы начать с пустой единицы компиляции и вызвать на ней метод AddUsings, как мы поступали ранее:

```
var unit = SyntaxFactory.CompilationUnit().AddUsings (usingDirective);
```

Добавить к такой единице компиляции определения типов можно путем их создания в аналогичной манере и вызова метода AddMembers:

```
// Создать простое пустое определение класса:
unit = unit.AddMembers (SyntaxFactory.ClassDeclaration ("Program"));
```

Наконец, дерево можно создать:

```
var tree = CSharpSyntaxTree.Create (unit.NormalizeWhitespace());
Console.WriteLine (tree.ToString());

// Вывод:
using System.Text;
class Program{}
```

### CSharpSyntaxRewriter

Для более сложных трансформаций синтаксического дерева можно создавать подклассы класса CSharpSyntaxRewriter.

Класс CSharpSyntaxRewriter похож на CSharpSyntaxWalker, который рассматривался ранее (см. раздел "CSharpSyntaxWalker"), за исключением того, что каждый метод Visit* принимает и возвращает синтаксический узел. За счет возвращения сущности, отличающейся от переданной, синтаксическое дерево можно "переписывать".

Например, следующий класс изменяет имена объявлений методов, представляя их символами верхнего регистра:

```
class MyRewriter : CSharpSyntaxRewriter
{
  public override SyntaxNode VisitMethodDeclaration
    (MethodDeclarationSyntax node)
  {
    // "Заменить" идентификатор метода его версией в верхнем регистре:
    return node.WithIdentifier (
      SyntaxFactory.Identifier (
        node.Identifier.LeadingTrivia,  // Сохранить старую дополнительную
                                        // синтаксическую информацию
        node.Identifier.Text.ToUpperInvariant(),
        node.Identifier.TrailingTrivia));  // Сохранить старую дополнительную
                                           // синтаксическую информацию
  }
}
```

Вот работать с классом MyRewriter:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() { Test(); }
  static void Test() {         }
}");

var rewriter = new MyRewriter();
var newRoot = rewriter.Visit (tree.GetRoot());
Console.WriteLine (newRoot.ToFullString());

// Вывод:
class Program
{
  static void MAIN() { Test(); }
  static void TEST() {        }
}
```

Обратите внимание, что вызов Test в главном методе не был переименован, потому что мы посещали только *объявления* членов, игнорируя *обращения*. Однако для надежного переименования обращений мы должны быть в состоянии определять, относятся ли вызовы метода Main или Test к типу Program, а не к какому-то другому типу. В такой ситуации одного лишь синтаксического дерева недостаточно; нам также нужна *семантическая модель*.

# Объекты компиляции и семантические модели

Объект компиляции заключает в себе синтаксические деревья, ссылки и параметры компиляции. Он служит двум целям:

- разрешить компиляцию в библиотеку либо исполняемый файл (фаза выпуска);
- открыть доступ к семантической модели, которая предлагает информацию о символах (полученную от связывания).

Семантическая модель необходима для реализации таких функциональных средств, как переименование символов или предоставление списков автозавершения кода в редакторе.

## Создание объекта компиляции

Независимо от того, заинтересованы вы в выдаче запросов к семантической модели или в проведении полной компиляции, первым шагом будет создание экземпляра CSharpCompilation с передачей конструктору (простого) имени сборки, подлежащей созданию:

```
var compilation = CSharpCompilation.Create ("test");
```

Простое имя сборки важно, даже если вы не планируете выпускать сборку, поскольку оно формирует часть удостоверения типов внутри объекта компиляции.

По умолчанию предполагается, что вы хотите создать библиотеку. Другой вид вывода (оконный исполняемый файл, консольный исполняемый файл и т.д.) можно указать следующим образом:

```
compilation = compilation.WithOptions (
  new CSharpCompilationOptions (OutputKind.ConsoleApplication));
```

Конструктор класса CSharpCompilationOptions имеет более десятка необязательных параметров, которые соответствуют опциям командной строки инструмента csc.exe. Таким образом, например, чтобы включить оптимизации компилятора и назначить сборке строгое имя, нужно поступить так:

```
compilation = compilation.WithOptions (
  new CSharpCompilationOptions (OutputKind.ConsoleApplication,
    cryptoKeyFile:"myKeyFile.snk",
    optimizationLevel:OptimizationLevel.Release));
```

Затем мы будем добавлять синтаксические деревья. Каждое синтаксическое дерево соответствует "файлу", который должен быть включен в объект компиляции:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() => System.Console.WriteLine (""Hello"");
}");

compilation = compilation.AddSyntaxTrees (tree);
```

Наконец, нам необходимо добавить ссылки. Простейшая программа потребует единственной ссылки на сборку mscorlib.dll, которую мы добавляем следующим образом:

```
compilation = compilation.AddReferences (
  MetadataReference.CreateFromFile (typeof (int).Assembly.Location));
```

Вызов метода MetadataReference.CreateFromFile приводит к чтению содержимого сборки в память, но без применения обычной рефлексии. Взамен метод использует высокопроизводительное переносимое средство чтения сборок (доступное в виде пакета NuGet) под названием System.Reflection.Metadata. Оно свободно от побочных эффектов и не загружает сборку в текущий домен приложения.

Экземпляр реализации PortableExecutableReference, получаемый из метода MetadataReference.CreateFromFile, может иметь значительный отпечаток в памяти, так что будьте осторожны и удерживайте только ссылки, которые нужны. Кроме того, если вы обнаружили, что неоднократно создаете ссылки на одну и ту же сборку, то полезно обдумать применение кеша (идеальным будет кеш, хранящий слабые ссылки).

Можно выполнить все действия за один шаг, вызвав перегруженную версию метода CSharpCompilation.Create, которая принимает синтаксические деревья, ссылки и параметры. Или же можно использовать текучий синтаксис в единственном выражении:

```
var compilation = CSharpCompilation.Create ("...")
  .WithOptions (...)
  .AddSyntaxTrees (...)
  .AddReferences (...);
```

## Диагностика

Компиляция может генерировать ошибки и предупреждения, даже если в синтаксических деревьях отсутствуют ошибки. Примером может быть недостающее импортирование пространства имен, опечатка при ссылке на имя типа или члена и неудавшееся выведение параметра типа. Для получения ошибок и предупреждений понадобится вызвать метод GetDiagnostics на объекте компиляции. В результат будут включены также и любые синтаксические ошибки.

## Выпуск сборки

Создание выходной сборки сводится просто к вызову метода Emit:

```
EmitResult result = compilation.Emit (@"c:\temp\test.exe");
Console.WriteLine (result.Success);
```

Если значением result.Success является false, тогда EmitResult имеет также свойство Diagnostics, предназначенное для указания на ошибки, которые произошли во время выпуска (сюда входит и диагностика из предшествующих этапов). Если метод Emit терпит неудачу из-за ошибки файлового ввода-вывода, то вместо передачи кодов ошибок он сгенерирует исключение.

Метод Emit также позволяет указывать путь к файлу .pdb (для отладочной информации) и путь к файлу XML-документации.

## Выдача запросов к семантической модели

Вызов метода GetSemanticModel на объекте компиляции возвращает *семантическую модель* для синтаксического дерева:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() => System.Console.WriteLine (123);
}");

var compilation = CSharpCompilation.Create ("test")
  .AddReferences (
    MetadataReference.CreateFromFile (typeof(int).Assembly.Location))
  .AddSyntaxTrees (tree);

SemanticModel model = compilation.GetSemanticModel (tree);
```

(Причина, по которой необходимо указывать дерево, связана с тем, что объект компиляции может содержать множество деревьев.)

Можно было бы предположить, что семантическая модель похожа на синтаксическое дерево, но с бо́льшим количеством свойств и методов и более детализированной структурой. Это не так; нет никакой всеобъемлющей DOM-модели, которая бы ассоциировалась с семантической моделью. Взамен вы располагаете набором методов, вызываемых для получения семантической информации о конкретной позиции или узле в синтаксическом дереве.

Вывод из сказанного — вы не можете "исследовать" семантическую модель подобно тому, как обошлись бы с синтаксическим деревом, и работа с ней довольно похожа на игру "20 вопросов": проблема заключается в том, чтобы отыскать правильные вопросы. Существует около 50 обычных и расширяющих методов; в настоящем разделе мы раскроем ряд наиболее распространенных методов, в частности, те, которые демонстрируют принципы использования семантической модели.

Продолжая предыдущий пример, мы могли бы запросить информацию о символах для идентификатора WriteLine следующим образом:

```
var writeLineNode = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "WriteLine").Parent;

SymbolInfo symbolInfo = model.GetSymbolInfo (writeLineNode);
Console.WriteLine (symbolInfo.Symbol);  // System.Console.WriteLine(int)
```

Структура SymbolInfo — это оболочка для символов, нюансы которой мы вскоре обсудим. Начнем с символов.

## Символы

В синтаксическом дереве имена, подобные"System", "Console" и "WriteLine", разбираются как *идентификаторы* (узел `IdentifierNameSyntax`). Идентификаторы мало что значат, и синтаксический анализатор не делает никакой работы для их "понимания", а только проводит различие между ними и контекстными ключевыми словами.

Семантическая модель также способна трансформировать идентификаторы в *символы*, которые имеют информацию о типе (выходные данные фазы *связывания*).

Все символы реализуют интерфейс `ISymbol`, хотя для каждого вида символов предусмотрен более специфичный интерфейс. В нашем примере "System", "Console" и "WriteLine" отображаются на символы следующих типов:

```
"System"      INamespaceSymbol
"Console"     INamedTypeSymbol
"WriteLine"   IMethodSymbol
```

Некоторые типы символов вроде `IMethodSymbol` имеют концептуальный аналог в пространстве имен `System.Reflection` (`MethodInfo` в данном случае); тогда как ряд других типов символов, таких как `INamespaceSymbol`, аналогов не имеют. Это объясняется тем, что система типов Roslyn существует для оказания помощи компилятору, в то время как система типов `Reflection` предназначена для содействия среде CLR (после того, как исходный код исчезает).

Тем не менее, работа с типами `ISymbol` во многом подобна применению API-интерфейса рефлексии, который был описан в главе 19. Расширим предыдущий пример:

```
ISymbol symbol = model.GetSymbolInfo (writeLineNode).Symbol;

Console.WriteLine (symbol.Name);                    // WriteLine
Console.WriteLine (symbol.Kind);                    // Method
Console.WriteLine (symbol.IsStatic);                // True
Console.WriteLine (symbol.ContainingType.Name);     // Console

var method = (IMethodSymbol) symbol;
Console.WriteLine (method.ReturnType.ToString());   // void
```

Вывод в последней строке кода иллюстрирует тонкое отличие от API-интерфейса рефлексии. Обратите внимание на то, что слово "void" представлено в нижнем регистре; это является спецификацией C# (API-интерфейс рефлексии безразличен к языкам). Подобным же образом вызов метода `ToString` на типе `INamedTypeSymbol` для `System.Int32` возвращает "int". Есть кое-что еще, чего не удастся добиться с помощью API-интерфейса рефлексии:

```
Console.WriteLine (symbol.Language);  // C#
```

Благодаря наличию API-интерфейса синтаксических деревьев классы, представляющие синтаксические узлы, отличаются для языков C# и VB (хотя они совместно используют абстрактный базовый тип `SyntaxNode`). Это имеет смысл, т.к. языки обладают разной лексической структурой. По контрасту `ISymbol` и производные от него интерфейсы разделяются между C# и VB. Однако их внутренние конкретные реализации *специфичны* для каждого языка, а вывод, получаемый из их методов, отражает отличия, характерные для того или иного языка.

Можно также выяснить, откуда поступил символ:

```
var location = symbol.Locations.First();
Console.WriteLine (location.Kind);                    // MetadataFile
Console.WriteLine (location.MetadataModule
                == compilation.References.Single()    // True
```

Если символ был определен в принадлежащем нам исходном коде (т.е. в синтаксическом дереве), тогда свойство `SourceTree` возвратит такое дерево, а свойство `SourceSpan` – его местоположение в дереве:

```
Console.WriteLine (location.SourceTree == null);      // True
Console.WriteLine (location.SourceSpan);              // [0..0)
```

Частичный тип может иметь множество определений и соответственно множество местоположений.

Приведенный ниже запрос возвращает все перегруженные версии метода `WriteLine`:

```
symbol.ContainingType.GetMembers ("WriteLine").OfType<IMethodSymbol>()
```

Можно также вызывать метод `ToDisplayParts` на символе. Он возвратит коллекцию "частей", которые образуют полное имя; в данном случае `System.Console.WriteLine(int)` включает четыре символа, отделяемые друг от друга знаками пунктуации.

### SymbolInfo

Если вы реализуете средство автозавершения кода для какого-то редактора, то вам понадобится получать символы для кода, который пока еще не завершен или некорректен. Например, взгляните на следующий незавершенный код:

```
System.Console.Writeline(
```

Поскольку метод `WriteLine` перегружен, сопоставление с единственной реализацией `ISymbol` невозможно. Взамен необходимо предложить пользователю варианты на выбор. Для такой цели в семантической модели имеется метод `GetSymbolInfo`, возвращающий структуру `ISymbolInfo`, которая содержит показанные далее свойства:

```
ISymbol Symbol
ImmutableArray<ISymbol> CandidateSymbols
CandidateReason CandidateReason
```

В случае ошибки или неоднозначности свойство `Symbol` возвращает `null`, а свойство `CandidateSymbols` – коллекцию с наилучшими совпадениями. Свойство `CandidateReason` возвращает перечисление, сообщающее о том, что именно пошло не так.

Чтобы получить информацию об ошибках и предупреждениях для раздела кода, можно также вызвать метод `GetDiagnostics` на семантической модели, указав ему структуру `TextSpan`. Вызов `GetDiagnostics` без аргументов эквивалентен вызову того же самого метода на объекте `CSharpCompilation`.

## Доступность символов

В интерфейсе `ISymbol` имеется свойство `DeclaredAccessibility`, которое указывает, является ли символ открытым, защищенным, внутренним и т.д. Тем не менее,

для определения доступности заданного символа в конкретной точке исходного кода этого недостаточно. Скажем, локальные переменные обладают лексически ограниченной областью видимости, а защищенные члены класса доступны в местах исходного кода, относящихся к внутренностям этого класса или его производных классов. Справиться с задачей поможет метод `IsAccessible` класса `SemanticModel`:

```
bool canAccess = model.IsAccessible (42, someSymbol);
```

Здесь метод `IsAccessible` возвращает `true`, если `someSymbol` может быть доступен по смещению 42 в исходном коде.

### Объявленные символы

Вызов метода `GetSymbolInfo` на объявлении типа или члена не приводит к получению каких-либо символов. Например, предположим, что требуется символ для метода `Main`:

```
var mainMethod = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "Main").Parent;

SymbolInfo symbolInfo = model.GetSymbolInfo (mainMethod);
Console.WriteLine (symbolInfo.Symbol == null);            // True
Console.WriteLine (symbolInfo.CandidateSymbols.Length);  // 0
```

Прием применим не только к объявлениям типов/членов, но и к любому узлу, где *вводится* новый символ, а не *потребляется* существующий.

Чтобы получить символ, необходимо взамен вызвать метод `GetDeclaredSymbol`:

```
ISymbol symbol = model.GetDeclaredSymbol (mainMethod);
```

В отличие от `GetSymbolInfo` метод `GetDeclaredSymbol` либо успешно завершается, либо нет. (Если он терпит неудачу, то из-за того, что не может найти допустимый узел объявления.)

В качестве еще одного примера рассмотрим метод `Main` следующего вида:

```
static void Main()
{
  int xyz = 123;
}
```

Тип `xyz` можно выяснить так:

```
SyntaxNode variableDecl = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "xyz").Parent;

var local = (ILocalSymbol) model.GetDeclaredSymbol (variableDecl);
Console.WriteLine (local.Type.ToString());           // int
Console.WriteLine (local.Type.BaseType.ToString()); // System.ValueType
```

### TypeInfo

Иногда нужна информация о типе выражения или литерала, для которого явные символы отсутствуют. Взгляните на следующий код:

```
var now = System.DateTime.Now;
System.Console.WriteLine (now - now);
```

Чтобы определить тип выражения `now - now`, мы вызываем метод `GetTypeInfo` на семантической модели:

```
SyntaxNode binaryExpr = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "-").Parent;
TypeInfo typeInfo = model.GetTypeInfo (binaryExpr);
```

Структура `TypeInfo` имеет два свойства, `Type` и `ConvertedType`. Последнее указывает тип после любых неявных преобразований:

```
Console.WriteLine (typeInfo.Type);              // System.TimeSpan
Console.WriteLine (typeInfo.ConvertedType);     // object
```

Поскольку метод `Console.WriteLine` перегружен для приема типа `object`, но не `TimeSpan`, произошло неявное преобразование в `object`, которое было подтверждено свойством `typeInfo.ConvertedType`.

### Поиск символов

Мощной возможностью семантической модели является средство запрашивания всех символов, находящихся в области видимости, для конкретного места исходного кода. Результатом будет основа для формирования списков IntelliSense, когда пользователю нужен перечень доступных символов. Чтобы получить такой список, необходимо просто вызвать метод `LookupSymbols`, передав ему желаемое смещение в исходном коде. Ниже представлен завершенный пример:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main()
  {
    int x = 123, y = 234;
  }
}");

CSharpCompilation compilation = CSharpCompilation.Create ("test")
  .AddReferences (
    MetadataReference.CreateFromFile (typeof(int).Assembly.Location))
  .AddSyntaxTrees (tree);
SemanticModel model = compilation.GetSemanticModel (tree);

// Найти доступные символы в начале 6-й строки:
int index = tree.GetText().Lines[5].Start;

foreach (ISymbol symbol in model.LookupSymbols (index))
  Console.WriteLine (symbol.ToString());
```

Вот результат:

```
y
x
Program.Main()
object.ToString()
object.Equals(object)
object.Equals(object, object)
object.ReferenceEquals(object, object)
object.GetHashCode()
object.GetType()
object.~Object()
object.MemberwiseClone()
Program
Microsoft
System
Windows
```

(В случае импортирования пространства имен `System` мы увидим сотни дополнительных символов для типов, определенных в этом пространстве имен.)

## Пример: переименование символа

Чтобы продемонстрировать рассмотренные средства, мы напишем метод переименования символа, который надежно работает в большинстве сценариев использования. В частности:

- символ может быть типом, членом, локальной переменной, переменной диапазона или переменной цикла;
- можно указывать символ либо в месте его применения, либо в точке его объявления;
- в случае класса или структуры будут переименованы статические конструкторы и конструкторы экземпляров;
- в случае класса будет переименован финализатор (деструктор).

Для краткости мы опустим некоторые проверки, такие как выяснение, не используется ли уже новое имя и не относится ли символ к краевому случаю, когда переименование потерпит неудачу. Наш метод будет принимать во внимание только одиночное синтаксическое дерево, поэтому он получит следующую сигнатуру:

```
public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
```

Одним очевидным способом реализации является создание подкласса класса `CSharpSyntaxRewriter`. Однако более элегантный и гибкий подход заключается в том, чтобы заставить метод `RenameSymbol` вызывать какой-то низкоуровневый метод, который возвращает тестовые промежутки, подлежащие переименованию:

```
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
```

Это позволяет редактору вызывать метод `GetRenameSpans` напрямую и применять только изменения (внутри транзакции отмены), избегая утери состояния редактора, что в противном случае может привести к замене всего текста.

В результате метод `RenameSymbol` становится относительно простой оболочкой вокруг `GetRenameSpans`. Мы можем использовать метод `WithChanges` класса `SourceText` для применения последовательности изменений в тексте:

```
public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
{
  IEnumerable<TextSpan> renameSpans = GetRenameSpans (model, token);

  SourceText newSourceText = model.SyntaxTree.GetText().WithChanges (
    renameSpans.Select (span => new TextChange (span, newName))
               .OrderBy (tc => tc));

  return model.SyntaxTree.WithChangedText (newSourceText);
}
```

Метод `WithChanges` генерирует исключение, если изменения не были упорядочены; именно потому мы вызываем метод `OrderBy`.

Теперь мы должны написать метод `GetRenameSpans`. Первым делом необходимо найти символ, соответствующий лексеме, которую мы хотим переименовать. Лексема

может быть частью либо объявления, либо употребления, так что мы сначала вызываем метод GetSymbolInfo, и если результатом окажется null, тогда вызываем метод GetDeclaredSymbol:

```
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
{
  var node = token.Parent;
  ISymbol symbol = model.GetSymbolInfo (node).Symbol
                ?? model.GetDeclaredSymbol (node);
  if (symbol == null) return null;  // Символ для переименования отсутствует.
```

Далее потребуется найти определения символа. Их можно получить из свойства Locations символа. (Учет множества местоположений обеспечивает надежную работу в сценарии с частичными классами и методами, хотя для того, чтобы данный пример стал пригодным в случае частичных классов, его следовало бы расширить работой с множеством синтаксических деревьев.)

```
var definitions =
  from location in symbol.Locations
  where location.SourceTree == node.SyntaxTree
  select location.SourceSpan;
```

Теперь необходимо найти случаи употребления символа. Мы начинаем с поиска лексем-потомков, имена которых совпадают с именем символа, т.к. это быстрый способ отсеять большинство лексем. Затем мы можем вызвать метод GetSymbolInfo на родительском узле лексемы и увидеть, соответствует ли он символу, который нужно переименовать:

```
var usages =
  from t in model.SyntaxTree.GetRoot().DescendantTokens()
  where t.Text == symbol.Name
  let s = model.GetSymbolInfo (t.Parent).Symbol
  where s == symbol
  select t.Span;
```

Операции, относящиеся к связыванию, такие как запрос информации о символе, имеют тенденцию выполняться медленнее операций, которые работают только с текстом или синтаксическими деревьями. Причина в том, что процесс связывания может требовать поиска типов в сборках, применения правил выведения типов и проверки расширяющих методов.

Если символ представляет собой что-то, отличающееся от именованного типа (является локальной переменной, переменной диапазона и т.п.), то работа сделана и можно возвращать определения вместе со случаями употребления:

```
if (symbol.Kind != SymbolKind.NamedType)
  return definitions.Concat (usages);
```

Если символ – именованный тип, тогда понадобится переименовать его конструкторы и деструктор при условии их наличия. Для этого мы организуем перечисление узлов-потомков в поиске объявлений типов с именами, совпадающими с именем типа, который подлежит переименованию. Затем мы получаем его *объявленный* символ, и если он совпадает с тем, который переименовывается, то мы находим его методы конструкторов и деструктора, возвращая промежутки текста его идентификаторов, когда они существуют:

```
  var structors =
    from type in model.SyntaxTree.GetRoot().DescendantNodes()
                                          .OfType<TypeDeclarationSyntax>()
    where type.Identifier.Text == symbol.Name
    let declaredSymbol = model.GetDeclaredSymbol (type)
    where declaredSymbol == symbol
    from method in type.Members
    let constructor = method as ConstructorDeclarationSyntax
    let destructor = method as DestructorDeclarationSyntax
    where constructor != null || destructor != null
    let identifier = constructor?.Identifier ?? destructor.Identifier
    select identifier.Span;

  return definitions.Concat (usages).Concat (structors);
}
```

Ниже показан завершенный код наряду с примерами его использования:

```
void Demo()
{
  var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static Program() {}
  public Program() {}
  static void Main()
  {
    Program p = new Program();
    p.Foo();
  }
  void Foo() => Bar();
  void Bar() => Foo();
}
");

  var compilation = CSharpCompilation.Create ("test")
    .AddReferences (
      MetadataReference.CreateFromFile (typeof(int).Assembly.Location))
    .AddSyntaxTrees (tree);

  var model = compilation.GetSemanticModel (tree);

  var tokens = tree.GetRoot().DescendantTokens();

  // Переименовать класс Program в Program2:
  SyntaxToken program = tokens.First (t => t.Text == "Program");
  Console.WriteLine (RenameSymbol (model, program, "Program2").ToString());

  // Переименовать метод Foo в Foo2:
  SyntaxToken foo = tokens.Last (t => t.Text == "Foo");
  Console.WriteLine (RenameSymbol (model, foo, "Foo2").ToString());

  // Переименовать локальную переменную p в p2:
  SyntaxToken p = tokens.Last (t => t.Text == "p");
  Console.WriteLine (RenameSymbol (model, p, "p2").ToString());
}
```

```
public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
{
  IEnumerable<TextSpan> renameSpans =
    GetRenameSpans (model, token).OrderBy (s => s);

  SourceText newSourceText = model.SyntaxTree.GetText().WithChanges (
    renameSpans.Select (s => new TextChange (s, newName)));

  return model.SyntaxTree.WithChangedText (newSourceText);
}

public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
{
  var node = token.Parent;

  ISymbol symbol =
    model.GetSymbolInfo (node).Symbol ??
    model.GetDeclaredSymbol (node);

  if (symbol == null) return null;  // Символ для переименования отсутствует.

  var definitions =
    from location in symbol.Locations
    where location.SourceTree == node.SyntaxTree
    select location.SourceSpan;

  var usages =
    from t in model.SyntaxTree.GetRoot().DescendantTokens ()
    where t.Text == symbol.Name
    let s = model.GetSymbolInfo (t.Parent).Symbol
    where s == symbol
    select t.Span;

  if (symbol.Kind != SymbolKind.NamedType)
    return definitions.Concat (usages);

  var structors =
    from type in model.SyntaxTree.GetRoot().DescendantNodes()
                                          .OfType<TypeDeclarationSyntax>()
    where type.Identifier.Text == symbol.Name
    let declaredSymbol = model.GetDeclaredSymbol (type)
    where declaredSymbol == symbol
    from method in type.Members
    let constructor = method as ConstructorDeclarationSyntax
    let destructor = method as DestructorDeclarationSyntax
    where constructor != null || destructor != null
    let identifier = constructor?.Identifier ?? destructor.Identifier
    select identifier.Span;

  return definitions.Concat (usages).Concat (structors);
}
```

# Предметный указатель

## Г



## Д



## Ж



## З

## И



## К

## Л



## М

## H



## O

## П

## P



## C

## Т



## У



## Ф

## Х



## Ц



## Ч



## Ш



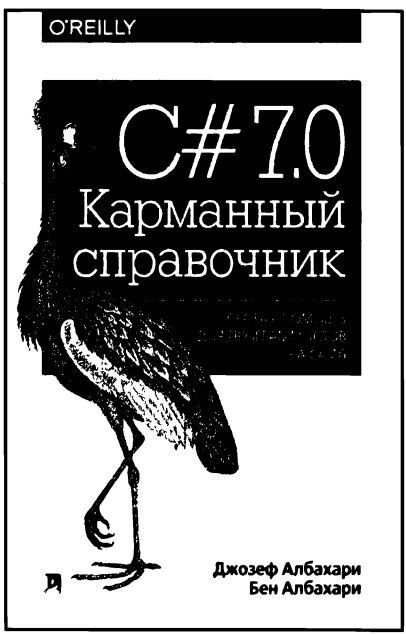
## Э



## Я

# C# 7.0
# КАРМАННЫЙ СПРАВОЧНИК

***Джозеф Албахари***
***Бен Албахари***




**www.williamspublishing.com**

Когда вам нужны ответы на вопросы по программированию на языке C# 7.0, этот узкоспециализированный справочник предложит именно то, что необходимо знать — безо всяких длинных введений или раздутых примеров. Легкое в чтении и идеальное в качестве краткого справочника, данное руководство поможет опытным программистам на C#, Java и C++ быстро ознакомиться с последней версией языка C#.

Эта книга написана авторами книги *C# 7.0. Справочник. Полное описание языка* и раскрывает все особенности языка C# 7.0.

- Фундаментальные основы C#
- Новые средства C# 7.0, включая кортежи, сопоставление по шаблону и деконструкторы
- Более сложные темы: перегрузка операций, ограничения типов, итераторы, типы, допускающие `null`, подъем операций, лямбда-выражения и замыкания
- Язык LINQ: последовательности, отложенное выполнение, стандартные операции запросов и выражения запросов
- Небезопасный код и указатели, специальные атрибуты, директивы препроцессора и XML-документация

**ISBN 978-5-9909446-1-9** **в продаже**

O'REILLY®

# C# 7.0 Справочник

Когда у вас возникают вопросы по языку C# 7.0 или среде CLR и основным сборкам .NET Framework, это ставшее бестселлером руководство предложит все необходимые ответы. С момента представления в 2000 году C# стал языком с замечательной гибкостью и широким размахом, но такое непрекращающееся развитие означает, что по-прежнему есть многие вещи, которые предстоит изучить.

Организованное вокруг концепций и сценариев использования, основательно обновленное седьмое издание книги снабдит программистов средней и высокой квалификации лаконичным планом получения знаний по C# и .NET. Погрузитесь в него и выясните, почему данное руководство считается исчерпывающим справочником по языку C#.

- Освойте должным образом все аспекты языка C#, от основ синтаксиса и переменных до таких сложных тем, как указатели и перегрузка операций
- Тщательно исследуйте LINQ с помощью трех глав, специально посвященных этой теме
- Узнайте о динамическом, асинхронном и параллельном программировании
- Научитесь работать с функциональными средствами .NET, в числе которых XML, взаимодействие с сетью, сериализация, рефлексия, безопасность, домены приложений и контракты кода
- Изучите доступный в C# 7.0 модульный компилятор как службу под названием Roslyn

"*C# 7.0 in a Nutshell* – одна из немногих книг, которые я держу на столе в качестве быстрого справочника."

Скотт Гатри,
Microsoft

"Как новички, так и эксперты найдут здесь все новейшие приемы программирования на C#."

Эрик Липперт,
C# MVP

**Джозеф Албахари** — соавтор книг *C# 6.0 in a Nutshell* (*C# 6.0. Справочник. Полное описание языка*, ИД "Вильямс", 2016 год), *C# 6.0 Pocket Reference* (*C# 6.0. Карманный справочник*, ИД "Вильямс", 2016 год), *C# 7.0 Pocket Reference* (*C# 7.0. Карманный справочник*, "Диалектика", 2017 год) и *LINQ Pocket Reference*, а также создатель LINQPad — популярной утилиты для подготовки кода и проверки запросов LINQ.

**Бен Албахари** — соавтор книги *C# 6.0 in a Nutshell* и бывший руководитель программы в команде разработчиков Entity Framework в Microsoft.

ISBN 978-5-6040043-7-1

9 785604 004371 18016

**Категория:** программирование
**Предмет рассмотрения:** C# / Microsoft .NET
**Уровень:** для пользователей средней и высокой квалификации

ДИАЛЕКТИКА

www.williamspublishing.com