

O'REILLY®

Программируем на C# 8.0

Разработка
приложений



Иэн Гриффитс

Programming C# 8.0

Build Cloud, Web, and Desktop Applications



Ian Griffiths

@CODELIBRARY_IT

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Программируем на C# 8.0

Разработка
приложений

Иэн Гриффитс



Санкт-Петербург · Москва · Минск

2021

ББК 32.973.2-018.1

УДК 004.43

Г85

Гриффитс Иэн

Г85 Программируем на C# 8.0. Разработка приложений. — СПб: Питер, 2021. — 944 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1638-6

C# —универсальный язык, который может практически всё! Иэн Гриффитс рассказывает о его возможностях с точки зрения разработчика, перед которым стоит задача быстро и эффективно создавать приложения любой сложности. Множество примеров кода научат работать с шаблонами, LINQ и асинхронными возможностями языка. Вы разберетесь с асинхронными потоками, ссылочными типами, допускающими значение NULL, сопоставлениями с образцом, реализациями по умолчанию для метода интерфейса, диапазонами и синтаксисом индексации и многим другим.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1

УДК 004.43

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

ISBN 978-1492056812 англ.

Authorized Russian translation of the English edition of Programming C# 8.0

ISBN 9781492056812 © 2020 Ian Griffiths

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same

© Перевод на русский язык ООО Издательство «Питер», 2021

© Издание на русском языке, оформление

ООО Издательство «Питер», 2021

© Серия «Бестселлеры O'Reilly», 2021

ISBN 978-5-4461-1638-6

Краткое содержание

Предисловие	12
Глава 1. Знакомство с языком C#	16
Глава 2. Основы написания кода на C#	55
Глава 3. Типы	141
Глава 4. Обобщения	247
Глава 5. Коллекции	271
Глава 6. Наследование	337
Глава 7. Время жизни объекта	379
Глава 8. Исключения	435
Глава 9. Делегаты, лямбды и события	470
Глава 10. LINQ	520
Глава 11. Реактивные расширения	590
Глава 12. Сборки	667
Глава 13. Отражение	701
Глава 14. Атрибуты	726
Глава 15. Файлы и потоки	755
Глава 16. Многопоточность	807
Глава 17. Асинхронные возможности языка	880
Глава 18. Эффективная работа с памятью	915
Об авторе	938
Об обложке	939

Оглавление

Предисловие	12
Для кого эта книга	12
Условные обозначения	12
Использование примеров кода	13
Благодарности.....	14
От издательства	15
Глава 1. Знакомство с языком C#	16
Почему C#?.....	17
Отличительные черты C#	19
Стандарты и реализация C#	23
Visual Studio и Visual Studio Code	30
Анатомия простой программы	34
Итог	54
Глава 2. Основы написания кода на C#	55
Локальные переменные	56
Инструкции и выражения	68
Комментарии и пробелы	77
Директивы препроцессора	80
Основные типы данных	86
Операторы	114
Управление потоком	121
Шаблоны.....	131
Итог	140
Глава 3. Типы	141
Классы	141
Структуры	162

Члены	175
Интерфейсы	231
Перечисления	236
Другие типы	240
Частичные типы и методы	244
Итог	245
Глава 4. Обобщения	247
Обобщенные типы	248
Ограничения	250
Нулевые значения	261
Обобщенные методы	263
Обобщения и кортежи	264
Внутренние обобщения	266
Итог	269
Глава 5. Коллекции	271
Массивы	271
Класс List<T>	292
Интерфейсы списков и последовательностей	295
Реализация списков и последовательностей	303
Обращение к элементам по индексу и синтаксис диапазона	311
Словари	321
Множества	327
Очереди и стеки	329
Связные списки	330
Параллельные коллекции	331
Неизменяемые коллекции	332
Итог	336
Глава 6. Наследование	337
Наследование и преобразования	339
Наследование интерфейса	343
Обобщения	344
System.Object	352

Доступность и наследование	354
Виртуальные методы	356
Запечатанные методы и классы.....	368
Доступ к членам базового класса	369
Наследование и конструирование	370
Специальные базовые типы	376
Итог	377
Глава 7. Время жизни объекта	379
Сборка мусора.....	380
Деструкторы и финализация	412
IDisposable	416
Упаковка	426
Итог	434
Глава 8. Исключения.....	435
Источники исключений.....	438
Обработка исключений	443
Выдача исключений	453
Типы исключений.....	460
Необработанные исключения	466
Итог	469
Глава 9. Делегаты, лямбды и события	470
Делегаты	471
Анонимные функции	491
События	507
Сравнение делегатов и интерфейсов	517
Итог	518
Глава 10. LINQ	520
Выражения запроса	521
Отложенное вычисление	532
LINQ, обобщения и IQueryable<T>	536
Стандартные операторы LINQ	538

Генерирование последовательностей	585
Другие реализации LINQ	586
Итог	588
Глава 11. Реактивные расширения.....	590
Ключевые интерфейсы	592
Публикация и подписка с использованием делегатов	604
Построители последовательностей.....	610
Запросы LINQ	613
Операторы запроса из Rx	627
Планировщики	641
Субъекты.....	646
Адаптация	650
Операции для работы со временем	658
Итог	666
Глава 12. Сборки	667
Анатомия сборки	668
Определение типа	673
Загрузка сборок	677
Имена сборок	687
Итог	700
Глава 13. Отражение	701
Типы отражения.....	702
Контексты отражения	722
Итог	725
Глава 14. Атрибуты	726
Применение атрибутов	726
Определение и использование атрибутов	747
Итог	753
Глава 15. Файлы и потоки.....	755
Класс Stream	756
Ориентированные на текст типы	768

Файлы и каталоги	779
Сериализация	794
Итог	806
Глава 16. Многопоточность	807
Потоки	807
Синхронизация	828
Задачи	857
Другие асинхронные шаблоны	873
Отмена	875
Параллелизм	876
Итог	878
Глава 17. Асинхронные возможности языка	880
Ключевые слова <code>async</code> и <code>await</code>	881
Шаблон <code>await</code>	899
Обработка ошибок	905
Итог	913
Глава 18. Эффективная работа с памятью	915
(Не) копируйте это	916
Представление последовательных элементов с помощью <code>Span<T></code>	920
Представление последовательных элементов с помощью <code>Memory<T></code>	925
<code>ReadOnlySequence<T></code>	926
Обработка потоков данных с помощью конвейеров	927
Итог	937
Об авторе	938
Об обложке	939

*Посвящаю эту книгу своей прекрасной жене Деборе
и чудесным дочерям Хейзел, Виктории и Лире.
Спасибо за то, что делаете мою жизнь ярче!*

Предисловие

Язык C# существует уже около двух десятилетий. Он неуклонно развивался и в плане возможностей, и в плане размера, но основные характеристики Microsoft всегда сохраняла без изменений. Каждая новая возможность должна идеально вписываться в состав предыдущих, улучшая язык, а не превращая его в несвязный набор различных функций.

Несмотря на то что C# по-прежнему остается довольно простым языком, о нем можно сказать гораздо больше, чем о его первом воплощении. Поскольку охват книги достаточно большой, от читателей ожидается определенный уровень технической подготовки.

Для кого эта книга

Книга написана для опытных разработчиков, каковым являюсь я сам. Я решил написать такую книгу, которую хотел бы прочитать сам, если бы уже знал какой-то язык и вдруг решил изучить C#. В то время как более ранние издания объясняли некоторые базовые понятия — классы, полиморфизм и коллекции, я предполагаю, что читатели уже знакомы с этим. И хотя первые главы все еще описывают, как в C# представлены эти общие идеи, основное внимание уделяется деталям, специфичным для C#.

Условные обозначения

В книге используются следующие обозначения:

Курсив

Используется для обозначения новых терминов.

Моноширинный шрифт

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных среды, операторов и ключевых слов.

Моноширинный жирный

Обозначает команды или другой текст, который должен вводиться пользователем.

Моноширинный курсив

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.



Так выделяются советы и предложения.



Так обозначаются советы, предложения и примечания общего характера.



Так обозначаются предупреждения и предостережения.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) доступны для загрузки по адресу http://oreil.ly/Programming_Csharp.

Если у вас имеется технический вопрос или обнаружилась проблема с использованием примеров кода, отправьте электронное письмо на адрес bookquestions@oreilly.com.

Эта книга существует для того, чтобы помочь вам выполнить свою работу. В целом вы можете использовать любой фрагмент кода из примеров в этой книге в собственных программах и документации. Вам не нужно обращаться к нам за разрешением, если только вы не воспроизводите значительную часть кода. Например, написание программы, которая использует несколько фрагментов кода из этой книги, не требует получения разрешений. В свою очередь, продажа или распространение примеров из книг O'Reilly требует разрешения. Чтобы ответить на вопрос, сославшись на эту книгу и при-

ведя листинг кода, разрешения не требуется. Включение значительного количества примеров кода из этой книги в документацию вашего продукта требует разрешения.

Если вы считаете, что использование примеров кода выходит за рамки добросовестного использования или требует дополнительного разрешения, свяжитесь с нами по адресу permissions@oreilly.com.

Благодарности

Большое спасибо научным редакторам книги: Стивену Тoubу (Stephen Toub), Говарду ван Ройджену (Howard van Rooijen) и Глину Гриффитсу (Glyn Griffiths). Также я хотел бы поблагодарить тех, кто вычитывал отдельные главы или как-то помогал или предоставлял информацию, которая сделала эту книгу лучше. Вот эти люди: Брайан Расмуссен (Brian Rasmussen), Эрик Липперт (Eric Lippert), Эндрю Кеннеди (Andrew Kennedy), Даниэль Синклер (Daniel Sinclair), Брайан Рэнделл (Brian Randell), Майк Вудринг (Mike Woodring), Майк Таулти (Mike Taulty), Мэри Джо Фоули (Mary Jo Foley), Барт Де Смет (Bart De Smet), Мэттью Адамс (Matthew Adams), Джесс Панни (Jess Panni), Джонатан Джордж (Jonathan George), Майк Лара (Mike Larah), Кармел Ив (Carmel Eve) и Эд Фриман (Ed Freeman). В частности, спасибо компании endjin за то, что позволили мне не только выделить рабочее время на написание этой книги, но и предоставили для этого все условия.

Спасибо всем в O'Reilly, чей труд помог этой книге родиться на свет. В частности, спасибо Корбину Коллинзу (Corbin Collins) за его поддержку и Тайлеру Ортману (Tyler Ortman) за поддержку в начале проекта. Также благодарю Кассандру Фуртадо (Cassandra Furtado), Дебору Бейкер (Deborah Baker), Рона Билодо (Ron Bilodeau), Ника Адамса (Nick Adams), Ребекку Демарест (Rebecca Demarest), Карен Монтгомери (Karen Montgomery) и Кристен Браун (Kristen Brown) за их помощь в завершении работы. Спасибо также Соне Сарубе (Sonia Saruba) и Кристине Эдвардс (Christina Edwards) за тщательное редактирование текста и такую же тщательную корректуру. Наконец, спасибо Джону Осборну (John Osborn) за то, что принял меня в ряды авторов O'Reilly, когда я писал свою первую книгу.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Знакомство с языком C#

Язык программирования C# (произносится как «си шарп») широко применяется для разработки во множестве областей, включая веб-сайты, облачные системы, интернет вещей, машинное обучение, классические приложения, встроенные контроллеры, мобильные приложения, игры и утилиты командной строки. Язык C# наряду со вспомогательной средой выполнения .NET, был в центре внимания разработчиков Windows на протяжении почти двух десятилетий, но в последние годы язык добрался и до других платформ. В июне 2016 года Microsoft выпустила .NET Core версии 1.0, кросс-платформенную версию .NET, включающую веб-приложения, микросервисы и консольные приложения, написанные на C#, для работы в macOS и Linux, а также в Windows.

Этот шаг в сторону других платформ Microsoft сделала на фоне своего интереса к разработке с использованием открытого исходного кода. На заре истории C# Microsoft тщательно следила за всем своим исходным кодом¹, но сегодня почти все, что связано с C#, разрабатывается открыто, причем приветствуется вклад разработчиков, не имеющих отношения к Microsoft. Новые предложения по возможностям языка публикуются на GitHub, что позволяет вовлекать сообщество на самых ранних этапах. В 2014 году для ускорения разработки проектов с открытым исходным кодом в мире .NET был создан .NET Foundation (<https://dotnetfoundation.org/>), и теперь многие из наиболее важных проектов Microsoft на C# и .NET находятся под управлением этого фонда (в дополнение ко многим проектам, не принадлежащим Microsoft). В их число входят компилятор Microsoft C# (<https://github.com/>

¹ Все это было верно и в отношении предыдущего кросс-платформенного предложения Microsoft, а именно .NET. В 2008 году Microsoft выпустила Silverlight 2.0, который позволял C# работать в браузерах на Windows и macOS. Silverlight проигрывал битву со всеми растущими возможностями и универсальным охватом HTML5 и JavaScript, и его закрытый исходный код никак не помогал его делу.

dotnet/roslyn), .NET Core (<https://github.com/dotnet/core>), а также среда выполнения, библиотека классов и инструменты для создания .NET-проектов.

Почему C#?

Хотя для C# существует множество сценариев использования, всегда есть и другие языки программирования. Почему же стоит выбрать среди них C#? Этот выбор зависит от того, что именно требуется сделать, а также от того, что вам нравится или не нравится в том или ином языке программирования. Я считаю, что C# предоставляет значительные возможности, гибкость и производительность и работает на достаточно высоком уровне абстракции, из-за чего не приходится тратить огромные усилия на мелкие детали, не связанные напрямую с задачей, которую призвана решать моя программа.

Большая часть моих C# обусловлена методами программирования, которые поддерживает язык. Например, в нем имеются объектно-ориентированные функции, обобщения и функциональное программирование. Он поддерживает как динамическую, так и статическую типизацию. Он обеспечивает мощные функции, ориентированные на списки и множества, благодаря языку интегрированных запросов (LINQ). Он имеет встроенную поддержку асинхронного программирования.

В последнее время C# приобрел гибкость в управлении памятью. Среда выполнения всегда предоставляла сборщик мусора (GC), освобождающий разработчиков от значительной части работы, связанной с освобождением памяти, которую программа больше не использует. GC — это распространенная функция в современных языках программирования, и хотя она является благом для большинства случаев, существуют некоторые особые сценарии, в которых ее влияние на производительность — это проблема. Поэтому в C# 7.2 (выпущен в 2017 году) добавлены различные функции, которые позволяют более явно управлять памятью, что дает возможность обменять простоту разработки на производительность во время выполнения, но все это без потери безопасности типов. Это дает возможность создавать на C# приложения, для которых критична производительность, что годами было прерогативой менее безопасных языков, таких как C и C++.

Конечно, языки программирования не живут в изоляции, и им нужны высококачественные библиотеки с широким спектром возможностей. Не-

которые изящные и академически прекрасные языки восхищают до тех пор, пока вы не захотите сделать что-то прозаическое, например подключиться к базе данных или указать, где хранить пользовательские настройки. Независимо от того, насколько мощный набор идиом предоставляет язык, он должен обеспечивать полный и удобный доступ к службам базовой платформы. В данном случае C# оказывается в выигрышном положении благодаря среде выполнения, библиотеке классов и масштабной поддержке сторонних библиотек.

.NET включает в себя как среду выполнения, так и библиотеку основных классов, которые используют программы на C#. Часть среды выполнения называется *общязыковой средой выполнения* (Common Language Runtime, CLR), потому что она поддерживает не только C#, но и любой другой язык .NET. Например, Microsoft также предлагает расширения Visual Basic, F# и .NET для C++. CLR имеет общую систему типов (CTS), которая допускает свободное взаимодействие кода, написанного на разных языках, что означает, что библиотеки .NET могут без проблем использоваться из любого языка .NET: F# может использовать библиотеки, написанные на C#, C# может использовать библиотеки Visual Basic и т. д.

Помимо среды выполнения имеется обширная библиотека классов, которая предоставляет обертки для многих функций базовой операционной системы (ОС), но также и значительный объем собственного функционала, такого как классы коллекций или обработка JSON.

Библиотека классов, встроенная в .NET, – это еще не все, ведь многие другие системы предоставляют свои собственные библиотеки .NET. Например, существуют объемные библиотеки, которые позволяют программам на C# использовать популярные облачные сервисы. Как и следовало ожидать, Microsoft предоставляет всеобъемлющие библиотеки .NET для работы со службами в рамках своей облачной платформы Azure. Аналогично Amazon предоставляет полнофункциональный набор разработки для использования с Amazon Web Services (AWS) из C# и других языков .NET. И библиотекам нет необходимости быть связанными с фреймворками. Существует большая экосистема библиотек .NET, как коммерческих, так и бесплатных с открытым кодом. В их число входят математические утилиты, библиотеки синтаксического анализа и компоненты пользовательского интерфейса (UI), и это лишь некоторые из них. Даже если вам не повезло и вам нужна функция ОС, у которой нет обертки в библиотеке .NET, C# предлагает различные механизмы для работы с другими видами API, такими как API в стиле C,

доступные в Win32, macOS и Linux, или API на основе объектной модели компонентов (COM) в Windows.

Наконец, благодаря тому что .NET существует уже около двух десятилетий, многие организации инвестировали значительные средства в технологии, созданные на этой платформе. Поэтому C# часто становится логичным выбором, когда речь идет о получении прибыли от этих инвестиций.

Таким образом, с C# мы получаем мощный набор абстракций, встроенных в язык, сильную среду выполнения и легкий доступ к огромному количеству функций библиотек и платформ.

Отличительные черты C#

Хотя внешне наиболее очевидной особенностью C# является свойственный семейству С синтаксис, возможно, более примечательно то, что он был первым языком, разработанным специально для использования в среде выполнения CLR. Как следует из названия, CLR достаточно гибок для поддержки множества языков. Однако есть важное различие между языком, который был расширен для поддержки CLR, и языком, который создан для работы в CLR. Это можно увидеть на примере расширений .NET в компиляторе C++ Microsoft: синтаксис использования этих функций заметно отличается от стандартного C++, что показывает четкое различие между собственно средой C++ и внешней средой CLR. Но даже отсутствие отличий в синтаксисе² не гарантирует разногласий, когда две среды будут работать по-разному. Например, если вам нужна коллекция чисел с динамическим изменением размера, то какой класс коллекции в C++ следует использовать: такой как `vector<int>` или один из .NET, например `List<int>?` Какой бы из них вы ни выбрали, он не всегда будет правильным решением: библиотеки C++ не будут знать, что делать с коллекцией .NET, тогда как API .NET не смогут использовать тип из C++.

C# поддерживает и среду выполнения .NET, и библиотеку классов, поэтому таких дилемм не возникает. В только что рассмотренном сценарии `List<int>`

² Первый набор Microsoft.NET-расширений для C++ больше напоминал обычный C++. Оказалось, что использование имеющегося синтаксиса для чего-то совершенно отличного от обычного C++ создавало путаницу, поэтому Microsoft отказалась от первой системы (Managed C++) в пользу более нового и самобытного синтаксиса, который называется C++/CLI.

вне конкуренции. При использовании библиотеки классов .NET не возникает никаких проблем, потому что она создана для той же среды, что и C#.

Первая версия C# явила модель программирования, тесно связанную с базовой моделью CLR. C# на протяжении многих лет добавлял собственные абстракции, но они разрабатывались так, чтобы наилучшим образом соответствовать CLR. Это наделяет C# особым духом. Это также означает, что, если вы хотите понять C#, следует разобраться в CLR и в том, как она исполняет код.

Управляемый код и CLR

Долгие годы самым распространенным способом работы компилятора были обработка исходного кода и создание выходных данных в форме, которая могла бы выполняться непосредственно центральным процессором компьютера. Компиляторы генерировали машинный код, т. е. серию инструкций в некоем двоичном формате, понятном процессору компьютера. Многие компиляторы до сих пор работают таким образом, а вот компилятор C# — нет. Вместо этого он использует модель, содержащую управляемый код.

При использовании управляемого кода компилятор не генерирует машинный код, который выполняет процессор. Вместо этого он создает форму двоичного кода, называемого *промежуточным языком* (intermediate language, IL). Сам же исполняемый двоичный файл обычно, хотя и не всегда, создается позже, во время выполнения. Использование IL позволяет использовать функции, которые трудно или даже невозможно реализовать в более традиционной модели.

Возможно, наиболее заметным преимуществом управляемой модели является то, что выходные данные компилятора не привязаны к конкретной архитектуре процессора. Вы можете написать .NET-компонент, который может работать на 32-битной архитектуре x86, которую ПК использовали десятилетиями, но будет так же хорошо работать и в более современном 64-битном ее обновлении (x64) и даже на совершенно иных платформах, таких как ARM. (Например, в .NET Core появилась возможность работы на устройствах на базе ARM, таких как Raspberry Pi.) С языком, который компилируется непосредственно в машинный код, вам потребовалось бы создавать различные бинарные файлы для каждого из них. Но с .NET вы можете скомпилировать один компонент, который может работать не только на любой из них, но и на платформах, которые не поддерживались на мо-

мент компиляции, при условии, если подходящая среда выполнения стала доступна в будущем. В целом любое улучшение в генерации кода CLR — будь то поддержка новых архитектур процессоров или просто повышение производительности для уже существующих — мгновенно идет на пользу всем языкам .NET. Например, более старые версии CLR не использовали преимущества расширений векторной обработки, доступные на современных процессорах x86 и x64, но в текущих версиях они часто используются при генерации кода для циклов. От этого выиграл любой код, исполняемый в текущих версиях .NET Core, включая тот, который был написан за годы до того, как было добавлено это усовершенствование.

Точный момент, когда CLR генерирует исполняемый машинный код, может варьировать. Как правило, в нем используется подход, называемый *JIT-компиляцией* (just-in-time), при котором каждая отдельная функция компилируется при первом запуске. Тем не менее это не является обязательным условием. Есть различные способы, которыми код .NET может быть скомпилирован заранее, *ahead of time* (AoT). Есть инструмент под названием NGen, который способен делать это уже после установки. Приложения магазина Windows, созданные для обобщенной платформы Windows (UWP), используют инструменты сборки .NET Native, которые делают это раньше, в рамках сборки. .NET Core 3.0 добавляет новый инструмент под названием crossgen, который позволяет любому приложению .NET Core (а не только приложениям UWP) использовать генерацию собственного кода во время сборки. Тем не менее генерация исполняемого кода все еще может происходить во время выполнения, даже когда вы используете эти инструменты³ — многоуровневая функция компиляции во время выполнения может решить динамически перекомпилировать метод, чтобы лучше оптимизировать его для текущей выполняемой задачи. (Это может быть сделано независимо от того, используете ли вы JIT или AoT.) Виртуализированная природа управляемого выполнения предназначена для того, чтобы сделать такие вещи прозрачными для вашего кода, хотя иногда все это может сказываться не только на производительности. Например, виртуализированное выполнение оставляет некоторую свободу в том, когда и как среда исполнения выполняет определенную работу по инициализации, и вы иногда можете наблюдать результаты ее оптимизации, приводящие к удивительным результатам.

³ Исключением является .NET Native: он не поддерживает JIT, поэтому там отсутствует многоуровневая компиляция.

Управляемый код обладает единой информацией о типах. Этого требуют форматы файлов, определяемые интерфейсом командной строки (CLI), потому что только так можно использовать определенные функции времени выполнения. Например, .NET предлагает различные службы автоматической сериализации, в которых объекты могут быть преобразованы в двоичные или текстовые представления их состояния, и эти представления позднее могут быть превращены в объекты, возможно, уже на другом компьютере. Такая служба опирается на полное и точное описание структуры объекта, что гарантированно присутствует в управляемом коде. Информация о типе может использоваться и другими способами. К примеру, платформы юнит-теста могут использовать ее для проверки кода в тестовом проекте и реализации всех написанных вами юнит-тестов. Опорой для этого служат службы отражения CLR, речь о которых пойдет в главе 13.

Хотя тесная связь C# со средой выполнения является одной из основных определяющих черт языка, она не единственная. В основе разработки C# лежит определенная философия.

Универсальность вместо специализации

C# оказывает предпочтение универсальным языковым возможностям, нежели специализированным. За прошедшие годы Microsoft несколько раз расширяла C#, и разработчиками языка всегда подразумевались конкретные сценарии для новых возможностей. Однако они всегда старались, чтобы каждый новый добавляемый элемент был полезен и за пределами этих основных сценариев.

Например, несколько лет назад Microsoft решила добавить в C# функции, которые добавляют в него интеграцию с базами данных. Появившаяся в результате технология Language Integrated Query (LINQ, описанная в главе 10), безусловно, на это нацелена, но Microsoft достигла этого, не добавляя в язык непосредственной поддержки доступа к данным. Вместо этого Microsoft представила ряд довольно разномастных возможностей. К ним относятся улучшенная поддержка идиом функционального программирования, возможность добавления новых методов к существующим типам без использования наследования, поддержка анонимных типов, возможность получения объектной модели, представляющей структуру выражения, и введение синтаксиса запроса. Последний пункт имеет очевидную связь с доступом к данным, но остальные не так просто соотнести с поставленной задачей. Тем не менее они могут использоваться совместно таким образом, который

значительно упрощает некоторые задачи доступа к данным. Но все функции полезны и сами по себе, поэтому помимо поддержки доступа к данным они предполагают гораздо более широкий диапазон сценариев. Например, эти дополнения (появившиеся в C# 3.0) значительно упростили обработку списков, множеств и других групп объектов, поскольку новые функции работают для коллекций любого типа, а не только для баз данных.

Примером такой философии универсальности стала языковая особенность, присутствующая в C# в качестве прототипа, который в конечном итоге не был реализован разработчиками окончательно. Эта функция позволила бы добавлять XML непосредственно в исходный код, встраивая выражения для вычисления значений определенных фрагментов контента прямо во время выполнения. Прототип компилировал это в код, который генерировал завершенный XML во время исполнения.

Microsoft Research обнародовала эту функцию, но в конечном итоге та не вошла в C#, хотя позже появилась в другом языке Microsoft .NET — Visual Basic, который также получил некоторые специализированные функции запросов для извлечения информации из документов XML. Внедренные выражения XML — это относительно узкоспециализированное средство, полезное только при создании документов XML. Что касается запросов к XML-документам, C# поддерживает эту функциональность с помощью обобщенных функций LINQ, что не требует каких-либо специфических для XML функций языка. Звезда XML склонилась к закату, когда эта языковая концепция была поставлена под вопрос из-за того, что многие ее функции узурпировал JSON (который, несомненно, будет заменен чем-то новым в ближайшие годы). Если бы встроенный XML в свое время попал в C#, то теперь он выглядел бы весьма анахроничной диковинкой.

Новые функции, добавленные в последующих версиях C#, применяющих то же правило. Например, функции деконструкции и сопоставления с шаблоном, добавленные в C# в версиях 7 и 8, направлены на то, чтобы облегчить жизнь с помощью едва уловимых, но полезных методов, без привязки к какой-то конкретной области применения.

Стандарты и реализация C#

Прежде чем приступить к рассмотрению реального кода, нужно выяснить, на какую реализацию C# и на какую среду выполнения ориентироваться. Есть спецификации, которые определяют язык и поведение во время выполнения

для всех реализаций C#, как описано во врезке «C#, CLR и стандарты». Это сделало возможным появление нескольких реализаций C# и среды выполнения. На момент написания книги наиболее широко распространены три из них: .NET Framework, .NET Core и Mono. Путаницу вносит то, что за всеми тремя стоит Microsoft, но стоит сказать, что изначально все планировалось не так.

C#, CLR И СТАНДАРТЫ

Орган по стандартизации ECMA опубликовал две ОС-независимые спецификации, которые эффективно определяют язык C# и среду выполнения: ECMA-334 – это спецификация языка C#, а ECMA-335 определяет общеязыковую инфраструктуру (Common Language Infrastructure, CLI), виртуальную среду, в которой работают программы на C# и других языках .NET. Версии этих документов также были опубликованы Международной организацией по стандартизации как ISO/IEC 23270:2018 и ISO/IEC 23271:2012 соответственно. Из-за числа «2018» может показаться, что спецификация C# более современна, чем она есть на самом деле: языковые стандарты ECMA и ISO соответствуют версии 5.0 C#. На момент написания книги ECMA работает над обновленной спецификацией языка. Но следует иметь в виду, что эти конкретные стандарты обычно на несколько лет отстают от современного положения дел. Хотя стандарт IEC CLI имеет еще более старую дату – 2012 (как и ECMA-335) – спецификации среды выполнения меняются реже, чем языки, поэтому спецификация CLI намного ближе к текущим реализациям, несмотря на названия, указывающие на обратное.

ECMA-335 определяет интерфейс командной строки, который включает в себя все поведение, требуемое от среды выполнения (например, CLR .NET или среды выполнения Mono), и многое другое. Он определяет не только поведение среды выполнения (которое он называет Virtual Execution System, или VES), но также формат файлов для исполняемых и библиотечных файлов, а также общую систему типов (Common Type System). Кроме того, он определяет подмножество CTS, которое языки, как ожидается, должны поддерживать, дабы гарантировать взаимодействие языков, называемое общеязыковой спецификацией (Common Language Specification, CLS).

Таким образом, можно сказать, что реализация CLI от Microsoft – это скорее .NET, нежели просто CLR, хотя .NET включает в себя множество дополнительных функций, не входящих в спецификацию CLI. (Например, библиотека классов, которую требует CLI, составляет лишь небольшое подмножество гораздо более крупной библиотеки .NET.) CLR фактически является VES для .NET, но вы вряд ли когда-либо увидите термин VES, используемый вне спецификации, почему в этой книге я в основном и говорю о CLR (или просто среде выполнения). Понятия CTS и CLS используются более широко, и я еще вернусь к ним в этой книге.

Проект Mono был запущен в 2001 году и изначально создавался не Microsoft. (Вот почему у него отсутствует .NET в названии — он может использовать название C#, потому что именно так стандарты именуют язык, а .NET — это торговая марка Microsoft.) Изначально перед Mono стояла задача позволить разработку приложений для Linux на C#, но позже добавилась поддержка iOS и Android. Этот важный шаг помог Mono найти свою нишу, поскольку теперь он в основном используется для создания кросс-платформенных приложений для мобильных устройств на C#. Это изначально был проект с открытым исходным кодом, который за время своего существования был поддержан различными компаниями. С 2011 года и до момента написания находится под управлением компании Xamarin. Microsoft приобрела Xamarin в 2016 году и на данный момент сохраняет его как отдельный бренд, позиционируя свою среду выполнения Mono как способ запуска кода C# на мобильных устройствах.

А что насчет двух других реализаций, в названии которых есть .NET?

Несколько .NET от Microsoft (временных)

В течение примерно семи лет всегда была только одна текущая версия .NET, но с 2008 года все стало усложняться. Сначала это было связано с профильными вариантами .NET, предназначенными для появляющихся и исчезающих платформ пользовательского интерфейса, в том числе Silverlight, несколькими вариантами Windows Phone, а также представленными в Windows 8 приложениями магазина (Store Applications). Хотя некоторые из них еще поддерживаются, все они тупиковые, за исключением приложений магазина, которые превратились в обобщенную платформу Windows (UWP). UWP перешел на .NET Core, из-за чего другие ветви .NET устарели.

Но даже игнорируя эти фактически несуществующие ответвления .NET, на момент написания этой книги Microsoft все еще выпускает две текущие версии .NET: .NET Framework (только для Windows, с закрытым исходным кодом) и .NET Core (кросс-платформенный, с открытым исходным кодом). В мае 2019 года Microsoft объявила, что намерена вернуться к единой текущей версии в ноябре 2020 года. В долгосрочной перспективе это уменьшит путаницу, а вот в ближайшей перспективе это лишь еще больше усложнит ситуацию, так как появится дополнительная версия, которую следует иметь в виду.

Один слегка озадачивающий аспект всего этого — незначительные различия в именах разных .NET. Первые 15 лет .NET Framework означал сочетание

двух вещей: среды выполнения и библиотеки классов. Его среда выполнения называлась CLR. Библиотека классов называлась различными именами, включая Base Class Library (BCL; вводящее в заблуждение имя, поскольку спецификации ECMA определяют термин «BCL» как нечто более узкое) или Framework Class Library.

Сегодня у нас есть еще и .NET Core. Его среда выполнения называется .NET Core Common Language Runtime (или просто CoreCLR), и это вполне однозначное имя: мы можем говорить о .NET Core CLR или .NET Framework CLR, и всегда очевидно, что мы имеем в виду. И на протяжении всей этой книги, когда я говорю о CLR или среде выполнения без каких-либо уточнений, это происходит потому, что речь идет о чем-то, что относится к обеим реализациям. К сожалению, .NET Core называет свою библиотеку классов .NET Core Framework (или CoreFX). Это не конструктивно, потому что еще до .NET Core слово *Framework* использовалось для обозначения комбинации CLR и библиотеки. И, чтобы еще больше осложнить ситуацию, многие в Microsoft теперь называют .NET Framework «desktop», чтобы дать понять, что речь не идет о .NET Core. (Это всегда сбивало с толку, потому что многие люди используют эту «desktop»-версию для серверных приложений. Более того, первый в истории выпуск .NET Core был предназначен для UWP, поддерживающего только приложения Windows. Прошел год, прежде чем Microsoft выпустила версию, которая способна на что-то большее⁴. И теперь, когда в .NET Core 3.0 на Windows добавлена поддержка двух платформ пользовательского интерфейса .NET для настольных ПК — Windows Presentation Foundation (WPF) и Windows Forms, — большинство новых приложений для настольных компьютеров будут ориентированы на .NET Core, а не так называемую .NET «desktop».) На всякий случай, если что-то не до конца ясно, в табл. 1.1 обобщается текущая ситуация.

Таблица 1.1. Названия компонентов .NET

Платформа	Среда выполнения	Библиотека классов
.NET Framework (она же .NET desktop)	.NET CLR	.NET Framework Class Library
.NET Core	.NET Core CLR	.NET Core Framework

⁴ Как ни странно, этот самый первый релиз с поддержкой UWP в 2015 году, по-видимому, так и не получил официального номера версии. Выпуск .NET Core 1.0 датируется июнем 2016 года, т. е. примерно через год.

В 2020 году, если Microsoft будет придерживаться своего замысла, все имена снова будут скорректированы, а .NET Core и .NET Framework будут заменены простым «.NET»⁵. На момент написания этой книги Microsoft не определилась с конкретными именами для соответствующих сред выполнения и библиотек.

Но до этого времени у нас есть две «текущие» версии, каждая из которых способна делать то, что другая не умеет, и именно поэтому обе поставляются одновременно. Платформа .NET Framework работает только в Windows, тогда как .NET Core поддерживает Windows, macOS и Linux. Хотя это делает .NET Framework менее используемым, это же означает, что он способен поддерживать некоторые специфичные для Windows функции. Например, есть раздел библиотеки классов .NET Framework, посвященный работе со службами синтеза и распознавания речи Windows. Это невозможно в .NET Core, поскольку он может работать в Linux, где эквивалентные функции либо не существуют, либо слишком различаются, чтобы быть представленными через один и тот же .NET API.

.NET, который должен появиться в 2020 году, по сути является следующей версией .NET Core, только с более умным названием. С .NET Core связана большая часть разработок .NET за последние несколько лет. .NET Framework по-прежнему полностью поддерживается, но уже начинает отставать. Например, версия 3.0 платформы веб-приложений Microsoft, ASP.NET Core, будет работать только на .NET Core, но уже не на .NET Framework. Таким образом, уход со сцены .NET Framework и повышение .NET Core до единственно верного .NET — это неизбежное завершение процесса, который продолжается уже несколько лет.

Ориентация на различные версии .NET посредством .NET Standard

Разнообразие сред выполнения, каждая из которых имеет свои собственные версии библиотек классов, представляет собой проблему для тех, кто хочет сделать свой код доступным для других разработчиков. На <http://nuget.org> имеется репозиторий пакетов для компонентов .NET, где Microsoft публикует все создаваемые ею библиотеки .NET, которые не встроены в сам .NET, и где большинство разработчиков .NET публикуют библиотеки, которыми хотели бы поделиться. Так какую же версию следует использовать вам? У этого вопроса две грани: есть не только конкретная реализация (.NET

⁵ 10 ноября 2020 года действительно была выпущена .NET 5.0. — Примеч. ред.

Core, .NET Framework, Mono), но и версия (например, .NET Core 2.2 или 3.0, .NET Framework 4.7.2 или 4.8). И есть более старые варианты .NET, такие как Windows Phone или Silverlight, — Microsoft по-прежнему поддерживает многие из них, включая постоянную поддержку через различные библиотеки в NuGet. Многие авторы популярных пакетов с открытым исходным кодом, распространяемых через NuGet, также поддерживают множество старых типов и версий платформы.

Первоначально люди справлялись с проблемой версий, создавая несколько вариантов своих библиотек. Когда вы распространяете библиотеки .NET через NuGet, то можете встраивать в пакет несколько наборов двоичных файлов, ориентированных на разные .NET. Тем не менее одна из основных проблем заключается в том, что по мере появления новых форм .NET существующие на протяжении многих лет библиотеки не будут работать во всех новых средах выполнения. Компонент, написанный для .NET Framework 4.0, будет работать на всех последующих версиях .NET Framework, но не на .NET Core. Даже если исходный код компонента полностью совместим с .NET Core, вам потребуется отдельная версия, скомпилированная для этой платформы. И если автор используемой вами библиотеки не добавил явную поддержку .NET Core, это помешает ее использовать. Это плохо для всех. Авторы компонентов встали перед необходимостью создавать новые варианты своих компонентов, и поскольку это зависит от того, есть ли у авторов желание и время на эту работу, пользователи сталкиваются с тем, что не все компоненты, которые они хотят применить, доступны на нужной им платформе.

Чтобы избежать этого, Microsoft представила .NET Standard, который определяет общие подмножества поверхности API библиотеки классов .NET. Если пакет NuGet нацелен, скажем, на .NET Standard 1.0, это гарантирует, что он сможет работать на .NET Framework версий 4.5 или новее, .NET Core 1.0 или новее или Mono 4.6 или новее. И что очень важно, в случае появления очередного варианта .NET существующие компоненты будут работать без изменений, даже если эта новая платформа еще не существовала на момент написания. Это истинно до тех пор, пока новый вариант поддерживает .NET Standard 1.0.

Для обеспечения самого широкого охвата библиотеки .NET, опубликованные в NuGet, должны ориентироваться на самую низкую версию .NET Standard из возможных. Версии с 1.1 по 1.6 постепенно добавили больше функциональности в обмен на поддержку меньшего диапазона платформ. (Например, если вы хотите использовать компонент .NET Standard 1.3

в .NET Framework, это должен быть .NET Framework 4.6 или более поздней версии.) .NET Standard 2.0 стал большим шагом вперед и послужил важной вехой в эволюции NET Standard: в соответствии с текущими планами Microsoft это будет самый большой номер версии, способной работать на .NET Framework. Версии .NET Framework, начиная с 4.7.2 и далее, полностью поддерживают его, но .NET Standard 2.1 не будет работать ни на одной версии .NET Framework ни сейчас, ни в будущем. Он будет работать на .NET Core 3.0 и .NET (т. е. на будущих версиях .NET Core).

Будущие версии среды исполнения Xamarin Mono также, вероятно, будут поддерживать его, но для классического .NET Framework это конец пути.

Что это значит для разработчиков на C#? Если вы пишете код, который никогда не будет использоваться за пределами конкретного проекта, то обычно ориентируетесь на последнюю версию .NET Core. Если нужна какая-то специфическая для Windows функция, которой там нет, можно нацелиться на .NET Framework и использовать любой пакет NuGet, предназначенный для .NET Standard, вплоть до версии 2.0 (включительно), что означает, что подавляющее большинство того, что есть на NuGet, будет вам доступно. Если вы пишете библиотеки, которыми хотите делиться, следует ориентироваться на .NET Standard. Инструменты разработки Microsoft по умолчанию используют .NET Standard 2.0 для новых библиотек классов, что является разумным выбором — вы можете открыть свою библиотеку для более широкой аудитории, перейдя на версию ниже, но сегодня версии .NET, поддерживающие .NET Standard 2.0, широко доступны, так что ориентироваться на более старые версии следует только тогда, когда необходима поддержка разработчиков, все еще использующих более старые .NET Frameworks. (Microsoft делает это в большинстве своих библиотек NuGet, но вам не обязательно придерживаться того же режима поддержки более старых версий.) Если вы хотите использовать какие-то более новые функции (например, типы с эффективным использованием памяти, описанные в главе 18), может потребоваться более свежая версия .NET Standard. В любом случае инструменты разработки гарантируют, что вы используете только те API, которые доступны в любой версии .NET Standard, о поддержке которой вы заявляете.

Microsoft предоставляет нечто большее, чем просто язык и различные среды выполнения со связанными с ними библиотеками классов. Существуют также среды разработки, которые могут помочь вам писать, тестировать, отлаживать и поддерживать код.

Visual Studio и Visual Studio Code

Microsoft предлагает три среды разработки для настольных систем: Visual Studio, Visual Studio для Mac и Visual Studio Code. Все три предоставляют базовые функции — текстовый редактор, инструменты сборки и отладчик, но Visual Studio имеет наиболее полную поддержку разработки приложений на C# независимо от того, будут ли эти приложения работать на Windows или на других платформах. Эта среда самая старая и существовала еще до появления C#. Она родом из времен, предшествовавших открытому исходному коду, так что она к нему так и не перешла. Имеется множество доступных редакций: от бесплатных до невероятно дорогих.

Visual Studio — это интегрированная среда разработки (IDE), поэтому в ней используется подход «все включено». В дополнение к полнофункциональному текстовому редактору она предлагает инструменты визуального редактирования пользовательского интерфейса. В ней имеется глубокая интеграция с системами контроля версий, такими как git, а также с онлайн-системами, предоставляющими репозитории исходников, отслеживание ошибок и другие функции управления жизненным циклом приложений (ALM), такие как GitHub и система Azure DevOps от Microsoft. Visual Studio предлагает встроенные средства мониторинга производительности и диагностики. Она имеет различные функции для работы с приложениями, разработанными и развернутыми для облачной платформы Microsoft Azure. Ее функция Live Share предлагает удобный способ совместной работы удаленных разработчиков, помогающий объединять и просматривать код. Она обладает самым обширным набором функций рефакторинга из трех описанных здесь сред.

В 2017 году Microsoft выпустила Visual Studio для Mac, и это не порт версии для Windows. Он вырос из продукта под названием Xamarin — среды разработки на Mac, предназначенный для создания мобильных приложений на C#, работающих в среде выполнения Mono. Изначально Xamarin был независимым продуктом, но когда, как говорилось ранее, Microsoft приобрела написавшую его компанию, то включила в него различные функции из версии Visual Studio для Windows, переведя его под бренд Visual Studio.

Visual Studio Code (часто сокращаемый до VS Code) был впервые выпущен в 2015 году. Это кросс-платформенный продукт с открытым исходным кодом, поддерживающий Linux, а также Windows и Mac. Он основан на платформе Electron и написан преимущественно на TypeScript. (Это означает, что на самом деле это одна и та же программа во всех операционных системах.)

VS Code – более легковесный продукт, чем Visual Studio: базовая установка VS Code умеет лишь немногим больше, чем просто редактировать текст. Однако по мере открытия файлов вы обнаружите загружаемые расширения, которые, если вы вдруг решите их установить, способны добавить поддержку C#, F#, TypeScript, PowerShell, Python и многих других языков. (Механизм расширения общедоступен, так что любой может опубликовать расширение.) Поэтому, хотя в своей первоначальной форме VS Code не является интегрированной средой разработки (IDE) и больше напоминает простой текстовый редактор, модель расширяемости превращает этот редактор в довольно мощный инструмент. Широкий спектр расширений привел к тому, что VS Code стал необычайно популярным за пределами мира языков Microsoft, и это, в свою очередь, способствовало еще большему росту ассортимента расширений.

Visual Studio предлагает самый простой путь начать работу с C# – не нужно устанавливать какие-либо расширения или изменять какую-либо конфигурацию для разработки и запуска. Поэтому начну с краткого введения в работу в Visual Studio.



Вы можете загрузить бесплатную версию Visual Studio под названием Visual Studio Community по ссылке: <https://www.visualstudio.com/>.

Любой значимый проект на C# будет иметь несколько файлов исходного кода, и в Visual Studio они будут принадлежать к проекту. Каждый проект на выходе создает один целевой объект. Цель сборки может быть простым файлом – например, проект на C# может иметь результатом исполняемый файл или библиотеку, – но некоторые проекты генерируют более сложные выходные данные. Например, некоторые типы проектов выполняют сборку веб-сайтов. Веб-сайт обычно содержит несколько файлов, но в совокупности эти файлы представляют собой единый объект, а именно один веб-сайт. Выходные данные каждого проекта будут развернуты как единое целое, даже если состоят из нескольких файлов.

Файлы проекта обычно имеют расширения, заканчивающиеся на `.proj`. Например, большинство проектов C# имеют расширение `.csproj`, в то время как проекты C++ используют `.vcxproj`. Если вы изучите эти файлы в текстовом редакторе, то обнаружите, что обычно они содержат XML. (Но не всегда так. Visual Studio является расширяемой, и каждый тип проекта определяется

системой управления проектами, которая может использовать любой нравящийся ей формат, но встроенные языки используют XML.) Эти файлы описывают содержимое проекта и определяют способ его сборки. Формат XML, который Visual Studio использует для файлов проектов на C#, может обрабатываться инструментом msbuild, а также инструментом командной строки dotnet, если вы установили .NET Core SDK, позволяющий создавать проекты из командной строки. VS Code тоже умеет работать с этими файлами.



Исполняемые файлы Windows обычно имеют расширение .exe, в то время как библиотеки используют .dll (историческое сокращение для *dynamic link library*). Однако .NET Core помещает весь сгенерированный код в файлы .dll. Начиная с .NET Core 3.0, он может генерировать исполняемый файл начальной загрузки (с расширением .exe в Windows), но тот просто запускает среду выполнения, после чего загружает .dll, содержащую основной скомпилированный вывод. .NET Framework компилирует приложение непосредственно в самозагружающийся .exe (без отдельного .dll). В любом случае единственное различие между основным скомпилированным выводом приложения и библиотеки состоит в том, что в первом указана точка входа приложения. Оба типа файлов способны экспортить функции, которые могут использоваться другими компонентами. Это два примера так называемых *файлов сборки*, речь о которых пойдет в главе 12.

Часто придется работать с группами проектов. Например, хорошей практикой является написание тестов для своего кода, но большую часть тестового кода не нужно развертывать как часть приложения, поэтому обычно автоматизированные тесты помещаются в отдельные проекты. Вы можете разделить код и по другим причинам. Возможно, система, которую вы создаете, имеет настольное приложение и веб-сайт и у вас есть общий код, который вы хотели бы использовать в обоих приложениях. В этом случае понадобится один проект, который создает библиотеку, содержащую общий код, другой, создающий приложение для настольного компьютера, и еще один — для создания веб-сайта плюс еще три проекта, содержащих юнит-тесты для каждого из основных проектов.

Visual Studio помогает работать с несколькими смежными проектами посредством того, что называется *решением*. Решение — это попросту набор проектов, и хотя они обычно связаны между собой, они не обязаны быть таковыми, т. е. решение по сути представляет собой контейнер. Вы можете

увидеть текущее загруженное решение и все его проекты в обозревателе решений (*Solution Explorer*) Visual Studio. Рисунок 1.1 показывает решение с двумя проектами. (Я использую Visual Studio 2019, которая на момент написания является последней версией.) *Solution Explorer* показывает древовидное представление, в котором можно развернуть каждый проект, чтобы просмотреть составляющие его файлы. Эта панель обычно открыта в правом верхнем углу Visual Studio, но ее можно скрыть или закрыть. Открыть ее можно с помощью *View → Solution Explorer*.

Visual Studio загружает проект, только если он является частью решения. Когда вы создаете новый проект, вы можете добавить его в существующее решение, но, если вы этого не сделаете, Visual Studio создаст для вас новое. Если вы попытаетесь открыть существующий файл проекта, Visual Studio попытается найти соответствующее решение, а если не найдет, то создаст его. Это связано с тем, что многие операции в Visual Studio ограничены текущим загруженным решением. Когда вы собираете свой код, то обычно собираете именно решение. Настройки конфигурации — *Debug* и *Release* — контролируются на уровне решения. Глобальный текстовый поиск работает по всем файлам решения.

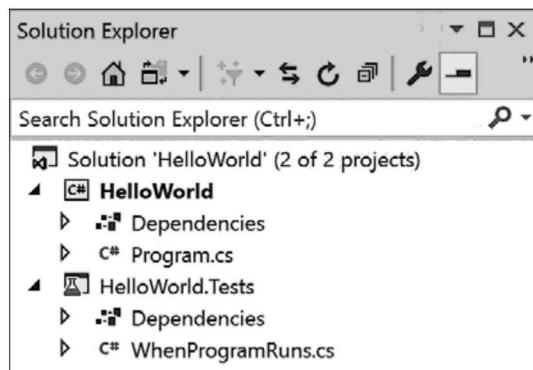


Рис. 1.1. Solution Explorer

Решение — это еще один текстовый файл, на этот раз с расширением `.sln`. Любопытно, что это не XML-файл — файлы решений используют свой собственный текстовый формат, хотя msbuild его понимает, равно как и VS Code. Если вы посмотрите на папку, содержащую ваше решение, то увидите папку `.vs`. (Visual Studio помечает ее как скрытую, но, если настроить проводник Windows для показа скрытых файлов, как это часто делают раз-

работчики, вы ее увидите.) Она содержит специфичные для пользователя настройки, например запись о том, какие файлы вы открыли и какой проект или проекты запускать при запуске сеансов отладки. Это гарантия того, что когда вы откроете проект, все будет более или менее на том же месте, где вы это оставили при последней работе над проектом. Поскольку эти настройки индивидуальны для каждого пользователя, папки `.vs` обычно не помещают в систему контроля версий.

Проект может принадлежать более чем одному решению. В большой базе исходного кода обычно имеется несколько файлов `.sln` с различными комбинациями проектов. Обычно у вас есть мастер-решение, содержащее каждый отдельный проект, но не все разработчики будут постоянно работать со всем кодом. Например, кто-то работающий с настольным приложением также захочет использовать совместно используемую библиотеку, но, скорее всего, не заинтересован в загрузке веб-проекта.

В рамках введения в язык я покажу, как создать новый проект и решение, а затем пройдусь по различным компонентам, которые Visual Studio добавляет в новый проект C#. Кроме того, я покажу, как добавить в решение проект юнит-теста.



Следующий раздел адресован разработчикам, не имеющим опыта работы в Visual Studio. Эта книга предназначена для опытных разработчиков, но не предполагает какого-либо предшествующего опыта работы с C# или Visual Studio. Если вы уже знакомы с работой в Visual Studio, то можете бегло просмотреть следующий раздел.

Анатомия простой программы

Если вы используете Visual Studio 2019, самый простой способ создать новый проект — использовать окно **Get Started**, которое открывается при запуске (рис. 1.2).

Если нажать кнопку **Create new project** в правом нижнем углу, откроется диалоговое окно нового проекта. Если Visual Studio уже запущена (или если вы используете более старую версию, которая не отображает окно **Get Started**), то в качестве альтернативы можете использовать пункт меню **File → New → Project** или, если предпочитаете хоткеи, нажмите **Ctrl+Shift+N**. Любое из этих действий открывает диалоговое окно **Create new project**, показанное на рис. 1.3.

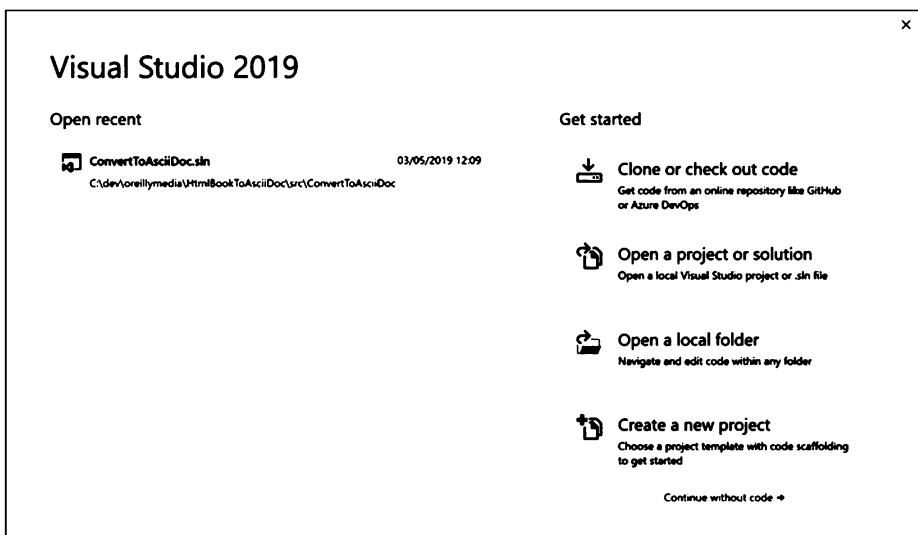


Рис. 1.2. Окно Get Started

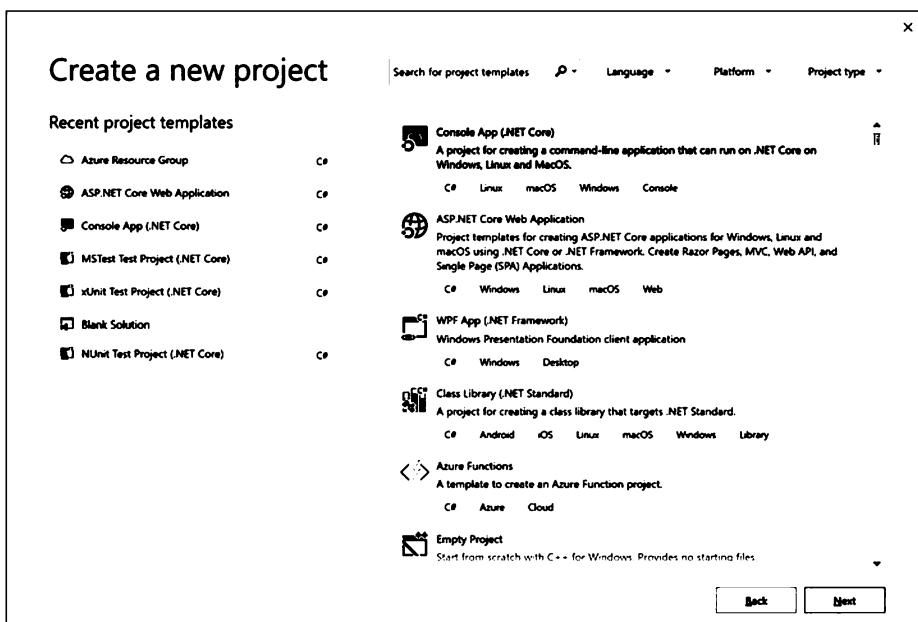


Рис. 1.3. Диалоговое окно Create new project

В этом окне предлагается на выбор список типов приложений. Точный набор будет зависеть от того, какую редакцию Visual Studio вы установили, а также от того, какие сценарии разработки приложений выбрали во время установки. Если вы установили хотя бы один из сценариев, включающих C#, то увидите вариант создания консольного приложения (.NET Core). Если вы выберете его и нажмете **Next**, увидите диалоговое окно **Configure your new project** (рис. 1.4).

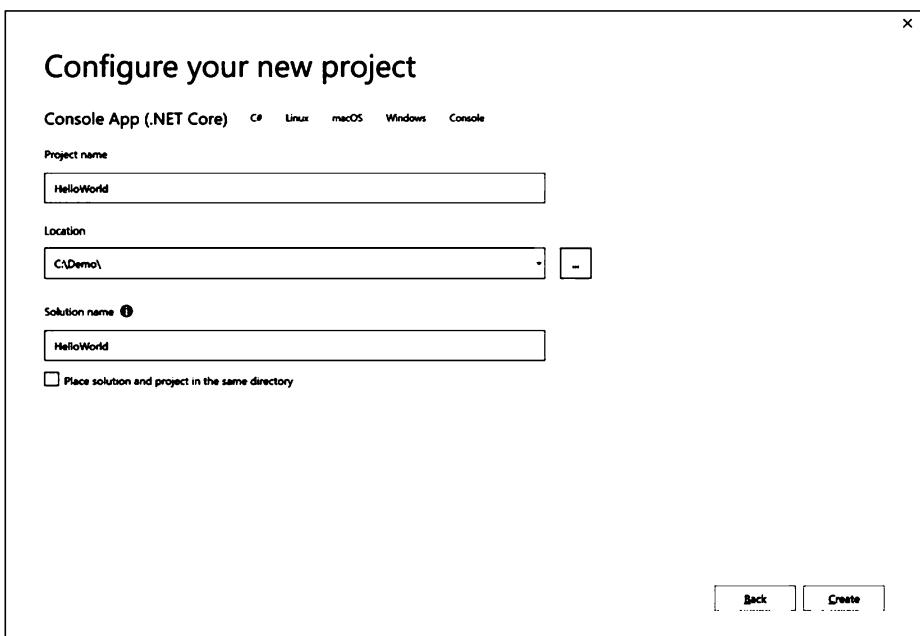


Рис. 1.4. Диалоговое окно **Configure your new project**

Так можно выбрать имя для вашего нового проекта, равно как и для содержащего его решения (по умолчанию используется то же имя). Вы также можете выбрать место расположения проекта на диске. Поле **Project name** отвечает за три вещи. Оно обуславливает имя файла .csproj на диске, определяет имя файла для скомпилированного вывода и устанавливает пространство имен по умолчанию для вновь создаваемого кода, что я объясню, когда мы перейдем к коду. (Все это можно изменить позже, если возникнет такое желание.)

В Visual Studio есть флажок **Place solution and project in the same directory**, который позволяет определить, как создается соответствующее решение. Если

вы отметите его, проект и решение будут иметь одно и то же имя и будут находиться в одной папке на диске. Если вы планируете добавить несколько проектов к вашему новому решению, вероятно, вам захочется, чтобы решение находилось в отдельной папке, а каждый проект сохранялся во вложенной папке. Если снять флагок, Visual Studio настроит все именно таким образом, дополнительно включив текстовое поле **Solution name**, чтобы при необходимости можно было задать имя, отличное от первого проекта. Помимо программы я собираюсь добавить к решению проект юнит-теста, поэтому я оставил флагок неотмеченным. Я назвал проект **HelloWorld**, а Visual Studio установил соответствующее имя решения, что меня более чем устраивает. Нажатие на **Create** создает новый проект на C#. Таким образом, сейчас у меня имеется решение с одним проектом внутри.

Добавление проекта в существующее решение

Чтобы добавить проект юнит-теста в решение, я могу перейти на панель **Solution Explorer**, щелкнуть правой кнопкой мыши узел решения (самый верхний) и выбрать **Add → New Project**. Откроется диалоговое окно, почти идентичное тому, что на рис. 1.3, но с заголовком **Add a new project**. Я хочу добавить проект тестирования. Можно просто прокрутить список типов проектов, но есть более быстрые способы. Я могу ввести **Test** в поле поиска в верхней части диалогового окна или нажать на кнопку **Project type** справа вверху, после чего выбрать **Test** из выпадающего списка. Любой вариант выдаст несколько разных типов тестовых проектов. Если вы видите проекты для языков, отличных от C#, нажмите кнопку **Language** рядом с окном поиска, чтобы оставить только относящиеся к C#. Но даже тогда вы увидите несколько типов проектов, потому что Visual Studio поддерживает несколько различных тестовых сред. Я выберу **MSTest Test Project (.NET Core)**.

При нажатии кнопки **Next** снова открывается диалоговое окно **Configure your new project**. Поскольку новый проект будет содержать тесты для моего проекта **HelloWorld**, я назову его **HelloWorld.Tests**. (Между прочим, подобное соглашение касательно имен не обязательно — я мог бы назвать его как угодно.) При нажатии **OK** Visual Studio создает второй проект, и оба они теперь представлены в обозревателе решений, который теперь выглядит примерно так, как показано на рис. 1.1.

Целью тестового проекта будет обеспечение того, чтобы основной проект работал так, как должен. Я предпочитаю стиль разработки, когда прежде

тестируемого кода пишутся тесты, поэтому мы начнем именно с тестового проекта. Чтобы иметь возможность выполнять свою работу, моему тестовому проекту потребуется доступ к коду в проекте `HelloWorld`. Visual Studio не пытается угадывать, какие проекты в решении могут зависеть от других проектов. Хотя их здесь только два, попытка угадать, скорее всего, дала бы неверный результат, потому что `HelloWorld` создаст исполняемую программу, в то время как проекты юнит-теста на выходе дают библиотеку. Наиболее очевидным предположением будет то, что программа будет зависеть от библиотеки, но у нас несколько необычное требование, заключающееся в том, что нашей библиотеке (которая на самом деле является тестовым проектом) необходим доступ к коду в нашем приложении.

Ссылка из одного проекта на другой

Чтобы сообщить Visual Studio о взаимосвязи этих двух проектов, я щелкаю правой кнопкой мыши узел `Dependencies` проекта `HelloWorld.Test` в `Solution Explorer` и выбираю пункт меню `Add Reference`. Откроется диалоговое окно `Reference Manager`, которое вы можете увидеть на рис. 1.5. Слева вы выбираете вид ссылки, который вам нужен, — в нашем случае я настраиваю ссылку на другой проект в том же решении, поэтому я развернул раздел `Projects` и выбрал `Solution`. В его середине перечислены все остальные проекты, но в данном случае там только один, поэтому я отмечаю элемент `HelloWorld` и нажимаю кнопку `OK`.

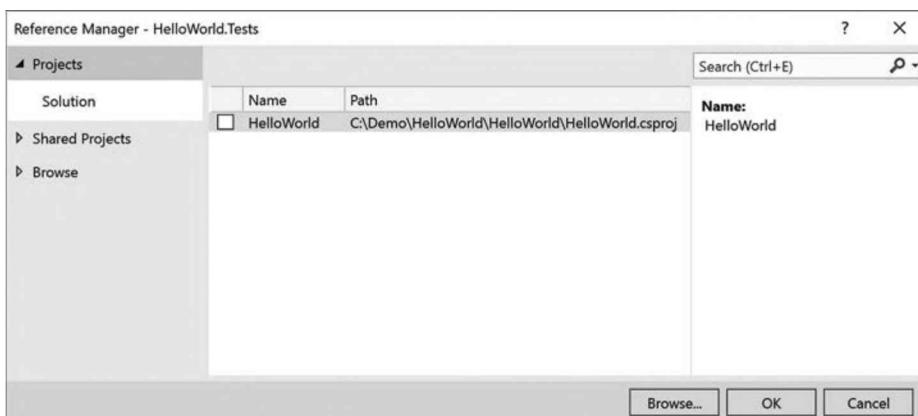


Рис. 1.5. Диалоговое окно Reference Manager

Ссылки на внешние библиотеки

Какой бы обширной ни была библиотека классов .NET, она не способна охватить все. Для .NET доступны тысячи полезных библиотек, многие из которых бесплатны. Microsoft производит все больше и больше библиотек отдельно от основной библиотеки классов .NET. Visual Studio поддерживает добавление ссылок с использованием системы NuGet, упомянутой ранее. Фактически наш листинг уже использует эту поддержку, хотя мы и выбрали собственную тестовую среду Microsoft «MSTest», которая не встроена в .NET. (Обычно вам не нужны сервисы юнит-теста во время выполнения, поэтому нет необходимости встраивать их в библиотеку классов, поставляемую вместе с платформой.) Если вы развернете узел **Dependencies** для проекта **HelloWorld**.Tests в **Solution Explorer** и затем развернете дочерний узел NuGet, то увидите различные пакеты NuGet, что отражено на рис. 1.6. (В вашем случае номера версий могут быть выше, так как эти библиотеки находятся в постоянном развитии.)

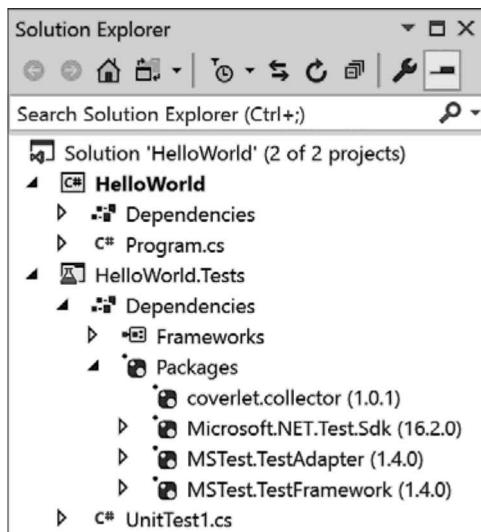


Рис. 1.6. Ссылки NuGet

Вы можете видеть четыре связанных с тестированием пакета, и все они добавлены в составе шаблона тестового проекта Visual Studio. NuGet — это система, основанная на пакетах, поэтому вместо добавления ссылки на одну

DLL вы добавляете ссылку на пакет, который может содержать несколько библиотек DLL и любые другие файлы, которые могут потребоваться для использования библиотеки.

В публичном хранилище пакетов, которое Microsoft развивает на веб-сайте <http://nuget.org>, хранятся копии всех библиотек, которые Microsoft не включает непосредственно в библиотеку классов .NET, но которые она при этом полностью поддерживает. (Используемая здесь среда тестирования является одним из примеров. Веб-платформа ASP.NET Core — это еще один пример.) Этот центральный репозиторий NuGet предназначен не только для нужд Microsoft. Любой может сделать свои пакеты доступными через этот сайт, поэтому именно здесь вы найдете подавляющее большинство бесплатных библиотек .NET.

Visual Studio способна выполнять поиск в основном репозитории NuGet. Если вы щелкнете правой кнопкой мыши по проекту или по его узлу Dependencies и выберете Manage NuGet Packages, он откроет окно диспетчера пакетов NuGet, как на рис. 1.7. Слева находится список пакетов из репозитория NuGet. Если вы выберете Installed вверху, то будут показаны только те пакеты, которые вы уже используете. Если вы нажмете кнопку Browse, то по умолчанию отобразятся популярные доступные пакеты, но также появится текстовое поле, с помощью которого вы сможете искать определенные библиотеки.

Также возможно создать свои собственные репозитории NuGet. Например, многие компании запускают репозитории за пределами своих брандмауэров, чтобы сделать разработанные в компании пакеты доступными для других сотрудников, не делая их публичными. Сайт <https://myget.org> специализируется на онлайн-хостинге, а частный хостинг пакетов является функцией Microsoft Azure DevOps, а также GitHub. Или вы можете просто разместить репозиторий в локальной файловой системе. Вы можете настроить NuGet для поиска среди любого количества репозиториев в дополнение к основному публичному.

Одна очень важная особенность пакетов NuGet заключается в том, что они способны содержать зависимости от других пакетов. Например, если вы посмотрите на пакет Microsoft.NET.Test.Sdk на рис. 1.6, маленький треугольник рядом с ним подскажет вам, что его древовидную структуру возможно раскрыть. Сделав это, вы увидите, что он зависит от некоторых других пакетов, включая Microsoft.CodeCoverage. Поскольку пакеты содержат собственные зависимости, Visual Studio может автоматически получать все необходимые вам пакеты.

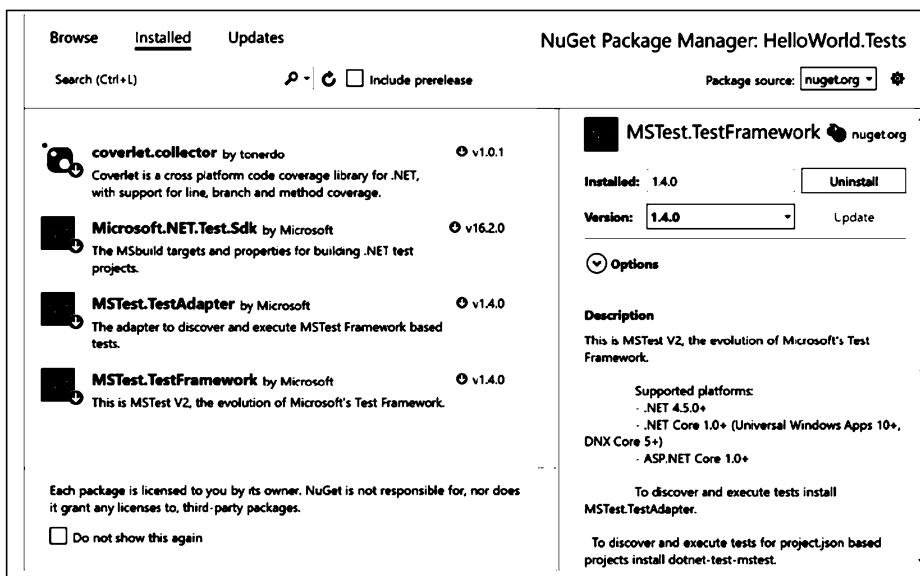


Рис. 1.7. NuGet Package Manager

Написание юнит-теста

Теперь нужно написать сам тест. Visual Studio предоставил мне тестовый класс для начала работы, и он содержится в файле `UnitTest1.cs`. Я хочу дать ему более информативное имя. Существуют различные точки зрения на то, как следует структурировать свои юнит-тесты. Некоторые разработчики рекомендуют один тестовый класс для каждого класса, который вы собираетесь протестировать. Мне же нравится стиль, в котором вы пишете класс для каждого сценария, в котором хотите протестировать определенный класс. В нем содержится по одному методу для каждого аспекта, который необходимо проверить в данном сценарии. Как вы, наверное, уже догадались из названий моих проектов, программа будет иметь лишь одно поведение: она будет при запуске отображать сообщение "Hello, world". Поэтому я переименую исходный файл `UnitTest1.cs` в `WhenProgramRuns.cs`. Этот тест должен проверить, что программа выводит необходимое сообщение при запуске. Сам по себе тест очень прост, но, к сожалению, до его запуска придется потрудиться. Листинг 1.1 показывает весь исходный файл; сам тест расположен ближе к концу и выделен жирным шрифтом.

Листинг 1.1. Юнит-тест для нашей первой программы

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HelloWorld.Tests
{
    [TestClass]
    public class WhenProgramRuns
    {
        private string _consoleOutput;

        [TestInitialize]
        public void Initialize()
        {
            var w = new System.IO.StringWriter();
            Console.SetOut(w);

            Program.Main(new string[0]);

            _consoleOutput = w.GetStringBuilder().ToString().Trim();
        }

        [TestMethod]
        public void SaysHelloWorld()
        {
            Assert.AreEqual("Hello, world!", _consoleOutput);
        }
    }
}
```

Расскажу обо всех особенностях этого файла, как только покажу саму программу. На данный момент наиболее интересной частью этого примера является метод `SaysHelloWorld`, который определяет ожидаемое поведение. Тест констатирует, что вывод программы — это сообщение "Hello, world". Если это не так, то тест сообщит о сбое. Приятно то, что сам по себе тест прост, однако код, который его настраивает, выглядит неуклюже. Проблема здесь в том, что обязательный первый пример, который демонстрируется практически во всех книгах по программированию, не очень подходит для юнит-теста отдельных классов или методов, потому что в реальности вы тестируете всю программу целиком. Мы хотим убедиться, что программа выводит в консоль конкретное сообщение. В реальном приложении можно было бы придумать какую-то абстракцию для вывода, и ваши юнит-тесты предоставили бы поддельную версию этой абстракции для целей тестирования. Но я хочу, чтобы мое приложение (которое тестируется в листинге 1.1)

соответствовало духу стандартного примера "Hello, world". Чтобы избежать чрезмерного усложнения основной программы, я реализовал перехват вывода в консоль, чтобы проверить, что программа отображает именно то, что от нее требуется. (Глава 15 опишет функции из пространства имен `System.IO`, которые я использую для достижения этой цели.)

Есть и вторая проблема. Исходя из названия обычно юнит-тест проверяет некоторую изолированную и небольшую часть программы. Но в этом случае программа настолько проста, что в ней есть лишь одна интересующая нас функция, и эта функция выполняется при запуске. Это означает, что мой тест должен будет вызвать точку входа в программу. Я мог бы запустить свою программу `HelloWorld` в совершенно новом процессе, но перехват ее вывода тогда был бы более сложным, чем когда он происходит внутри процесса, как показано в листинге 1.1. Вместо этого я просто напрямуюзываю точку входа в программу. В приложении C# точкой входа обычно является метод с именем `Main`, определенный в классе с именем `Program`. В листинге 1.2 показана соответствующая строка из листинга 1.1, которая передает пустой массив для имитации запуска программы без аргументов командной строки.

Листинг 1.2. Вызов метода

```
Program.Main(new string[0]);
```

К сожалению, здесь нас ждет очередная проблема. Точка входа в программу, как правило, доступна только для среды выполнения, так как это часть реализации вашей программы и делать ее публичной обычно нет нужды. Но я сделаю исключение, потому что именно там и будет работать код нашего примера. Итак, чтобы получить код для компиляции, нужно внести изменения в нашу основную программу. Листинг 1.3 показывает соответствующий код из файла `Program.cs` в проекте `HelloWorld`. (Очень скоро я покажу его целиком.)

Листинг 1.3. Делаем точку входа в программу доступной

```
public class Program
{
    public static void Main(string[] args)
    {
        ...
    }
}
```

Я добавил ключевое слово `public` в начало двух строк, чтобы сделать код доступным для теста, что позволило скомпилировать листинг 1.1. Есть и другие

способы, которыми этого можно добиться. Я мог бы оставить класс как есть, сделать метод внутренним, а затем применить `InternalsVisibleToAttribute` к моей программе, чтобы предоставить доступ только для набора тестов. Но атрибуты внутренней защиты и уровня сборки являются темами для последующих глав (главы 3 и 14 соответственно), поэтому для первого примера я решил действовать самым простым способом. Альтернативный подход покажу в главе 14.

Итак, я готов запустить свой тест. Для этого я открываю панель Unit Test Explorer в Visual Studio с помощью пункта меню `Test → Windows → Test Explorer`. Далее я собираю проект с помощью меню `Build → Build Solution`. После этого в модуле Unit Test Explorer отобразится список всех юнит-тестов, определенных в решении. Как вы можете видеть на рис. 1.8, он успешно обнаружил мой тест `SaysHelloWorld`. Нажатие на кнопку `Run all` (двойная стрелка в левом верхнем углу) запускает тест, но безуспешно, потому что до сих пор мы писали только тест, но ничего не сделали для нашей основной программы. Вы можете увидеть ошибку внизу рис. 1.8. Она говорит, что ожидалось сообщение "`Hello, world`", но фактический вывод в консоль был другим. (Надо признать, это не совсем так, потому что Visual Studio фактически добавила код в мое консольное приложение, которое выводит сообщение. Но в нем нет запятой, которой требует мой тест, а у буквы `w` неправильный регистр.)

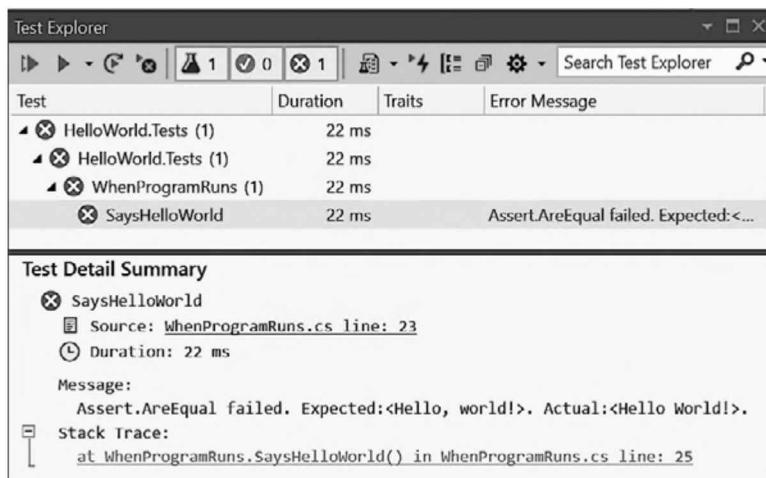


Рис. 1.8. Unit Test Explorer

Настало время взглянуть на нашу программу `HelloWorld` и внести исправления в код. Когда я создавал проект, Visual Studio сгенерировал различные файлы, в том числе `Program.cs`, который содержит точку входа в программу. Листинг 1.4 демонстрирует этот файл, включая изменения, которые я сделал в листинге 1.3. Я последовательно объясню каждый элемент, и это послужит полезным введением в некоторые важные элементы синтаксиса и структуры C#.

Файл начинается с *директивы using*. Она не является обязательной, но почти все исходные файлы содержат одну или несколько. Они сообщают компилятору, какие пространства имен мы хотели бы использовать, что поднимает очевидный вопрос: а что такое пространство имен?

Листинг 1.4. Program.cs

```
using System;

namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Пространства имен

Пространства имен привносят порядок и структуру в то, что в противном случае превратилось бы в жуткую свалку. Библиотека классов .NET содержит большое количество классов, еще больше их в сторонних библиотеках, не говоря уже о классах, которые вы напишете сами. Есть две проблемы, которые могут возникнуть при работе с таким количеством именованных объектов. Во-первых, достаточно трудно гарантировать уникальность имен, если только не давать классам сверхдлинные имена и не добавлять в них фрагменты со случайной тарабарщиной. Во-вторых, может оказаться непросто найти необходимый API; если вы не знаете или не можете угадать правильное имя, трудно найти то, что нужно в неструктурированном списке из многих тысяч элементов. Пространства имен решают обе эти проблемы.

Большинство типов .NET определены в пространстве имен. Типы, поставляемые Microsoft, имеют особенные пространства имен. Когда типы являются частью .NET, содержащиеся в них пространства имен начинаются с `System`, а когда они являются частью какой-либо технологии Microsoft, которая не является основной частью .NET, они обычно начинаются с `Microsoft`. Библиотеки других провайдеров обычно начинают с названия компании, в то время как библиотеки с открытым исходным кодом часто используют название своего проекта. Вы не обязаны помещать свои собственные типы в пространства имен, но делать это крайне рекомендуется. C# не рассматривает `System` как специальное пространство имен, поэтому ничто не мешает вам использовать его для своих собственных типов. Тем не менее если вы не вносите свой вклад в библиотеку классов .NET и собираетесь отправлять запрос на включение изменений в <https://github.com/dotnet/corefx>, тогда это плохая идея, потому что она может запутать других разработчиков. Для своего кода следует выбрать что-то более характерное, например название вашей компании или проекта.

Пространство имен обычно дает представление о предназначении типа. Например, все типы, относящиеся к обработке файлов, можно найти в пространстве имен `System.IO`, а те, что связаны с сетью, находятся в `System.Net`. Пространства имен могут образовывать иерархию. Таким образом, пространство имен `System` не только содержит типы. Оно также содержит и другие пространства имен, такие как `System.Net`, которые, в свою очередь, содержат еще больше пространств имен, таких как `System.Net.Sockets` и `System.Net.Mail`. Эти примеры показывают, что пространства имен действуют как своего рода описание, которое может помочь вам ориентироваться в библиотеке. Например, если вы ищете методы обработки регулярных выражений, то в доступных пространствах имен вы можете заметить `System.Text`. Взглянув туда, вы найдете пространство имен `System.Text.RegularExpressions`, и в этот момент вы будете уже достаточно уверены, что ищете в нужном месте.

Пространства имен также обеспечивают уникальность. Пространство имен, в котором определен тип, является частью полного имени этого типа. Это позволяет библиотекам использовать короткие и простые имена для сущностей. Например, API регулярного выражения включает в себя класс `Capture`, который представляет результаты захвата регулярного выражения. Если вам знаком софт, который работает с изображениями, термин «захват» чаще используется, когда речь идет о получении изображений, и вы можете решить, что `Capture` является наиболее подходящим именем для

класса в вашем собственном коде. Было бы досадно выбирать другое имя только потому, что наилучшее имя уже занято, особенно если в вашем коде получения изображения не используются регулярные выражения, а это означает, что вы даже не планировали использовать соответствующий тип.

Но на самом деле все хорошо. Оба типа могут называться `Capture` и при этом по-прежнему иметь разные имена. Полным именем класса `Capture` для работы с регулярными выражениями на самом деле будет `System.Text.RegularExpressions.Capture`, тогда как полное имя вашего класса будет включать в себя содержащее его пространство имен (например, `SpiffingSoftworks.Imaging.Capture`).

Если вы очень хотите, то можете указывать полное имя типа каждый раз, когда используете его, но большинство разработчиков не любят напрягаться, и им в помощь приходит директива `using` из начала листинга 1.4. Хотя в этом простом примере она всего только одна, часто в этом месте можно увидеть целый список директив. Они определяют пространства имен типов, которые исходный файл намеревается использовать. Обычно этот список редактируется в соответствии с требованиями вашего файла. В нашем примере Visual Studio добавила `using System` в момент создания проекта. В разных обстоятельствах она выбирает разные наборы директив. Например, если вы добавите класс, представляющий собой элемент пользовательского интерфейса, Visual Studio включит в список различные пространства имен, связанные с пользовательским интерфейсом.



Ранее вы видели, что в `References` проекта описано, какие библиотеки он использует. Вы можете подумать, что ссылки избыточны — неужели компилятор не способен сам определить из пространств имен, какие внешние библиотеки мы используем? Это было бы возможно при условии прямого соответствия между пространствами имен и библиотеками или пакетами, но это далеко не так. Иногда возникает очевидная связь — например, популярный пакет `Newtonsoft.Json` NuGet содержит файл `Newtonsoft.Json.dll`, который, в свою очередь, содержит классы в пространстве имен `Newtonsoft.Json`. Но часто такой связи нет — версия библиотеки классов .NET Framework содержит файл `System.Core.dll`, но пространства имен `System.Core` не существует. Поэтому компилятору необходимо указать, от каких библиотек зависит ваш проект, а также какие пространства имен использует тот или иной файл исходного кода. Более подробно мы рассмотрим сущность и структуру библиотечных файлов в главе 12.

Применение `using` позволяет вам использовать для класса краткое имя без дополнительного определения. Стока кода, которая позволяет моему примеру `HelloWorld` выполнять свою работу, использует класс `System.Console`, но из-за первой директивы `using` я могу называть ее просто `Console`. Фактически это единственный класс, который я намерен использовать, поэтому нет необходимости добавлять какие-либо другие директивы `using` в основную программу.

Даже с пространствами имен существует вероятность неоднозначности. Вы можете использовать два пространства имен, каждое из которых определяет класс с одним и тем же именем. Для использования этого класса вам будет необходимо явно указать его полное имя. Если в файле вам требуется часто использовать такие классы, вы все равно можете сэкономить при наборе текста: использовать полное имя только однажды, при определении псевдонима. Листинг 1.5 использует псевдонимы для разрешения конфликта, с которым я сталкивался не раз: платформа пользовательского интерфейса .NET, Windows Presentation Foundation (WPF), определяет класс `Path` для работы с кривыми Безье, многоугольниками и другими фигурами. Однако существует также и класс `Path` для работы с путями файловой системы, и вы можете одновременно использовать оба типа для создания графического представления содержимого файла. Простое добавление директив `using` для обоих пространств имен сделало бы простое имя `Path` неоднозначным, если использовать его без дополнительного определения. Но, как демонстрируется в листинге 1.5, возможно определить различимые псевдонимы для каждого из них.

Листинг 1.5. Устранение неоднозначности с помощью псевдонимов

```
using System.IO;
using System.Windows.Shapes;
using IoPath = System.IO.Path;
using WpfPath = System.Windows.Shapes.Path;
```

Используя псевдонимы, вы можете назначить `IoPath` в качестве синонима для класса `Path`, работающего с файлами, а `WpfPath` — для работающего с графикой.

Возвращаясь к примеру `HelloWorld`: сразу после директив `using` идет объявление пространства имен. В то время как с помощью директив объявляется, какие пространства имен будет использовать наш код, в объявлении пространства имен указывается пространство имен, в котором существует наш

код. Листинг 1.6 показывает соответствующий код из листинга 1.4. За ним следует открывающая скобка ({). Все между этой и закрывающей скобкой в конце файла будет находиться в пространстве имен `HelloWorld`. Кстати, вы можете ссылаться на типы в своем собственном пространстве имен без необходимости использования директивы `using`. Вот почему тестовый код в листинге 1.1 не включает в себя директивы `using HelloWorld;` — у него неявно есть доступ к этому пространству имен, потому что его код находится внутри объявленного пространства имен `HelloWorld.Tests`.

Листинг 1.6. Объявление пространства имен

```
namespace HelloWorld  
{
```

Visual Studio добавляет объявление пространства имен с тем же именем, что и ваш проект, в файлах исходного кода, которые она добавляет при создании нового проекта. От вас не требуется сохранять его неизменным — проект может содержать любую комбинацию пространств имен, а вы можете редактировать объявление пространства имен. Но если по всему своему проекту вы хотите систематически использовать что-то кроме имени проекта, следует указать на это Visual Studio, потому что это сгенерированное объявление получит не только первый файл `Program.cs`. По умолчанию Visual Studio добавляет объявление пространства имен на основе имени вашего проекта каждый раз, когда вы добавляете новый файл. Вы можете попросить ее использовать другое пространство имен для всех новых файлов, отредактировав свойства проекта. Если щелкнуть правой кнопкой мыши на узел проекта в `Solution Explorer` и выбрать `Properties`, то откроются свойства проекта, а если перейти на вкладку `Application` — появится текстовое поле `Default namespace`. Все, что вы туда впишете, будет использовано для объявлений пространства имен во всех новых файлах. (Это не изменит уже существующие файлы.) Это изменение добавляется свойство `<RootNamespace>` в файл `.csproj`.

Вложенные пространства имен

Как вы уже видели, библиотека классов .NET содержит вложенные пространства имен, и иногда довольно обширные. Если вы не создаете нечто большее, чем тривиальный пример, то обычно производите свои собственные вложенные пространства имен. Есть два способа сделать это. Например, вкладывать объявления пространства имен, как показано в листинге 1.7.

Листинг 1.7. Вкладывание объявлений пространства имен

```
namespace MyApp
{
    namespace Storage
    {
        ...
    }
}
```

Кроме того, вы можете просто указать полное пространство имен в одном объявлении, как показано в листинге 1.8. Это наиболее часто используемый стиль.

Листинг 1.8. Вложенное пространство имен одним объявлением

```
namespace MyApp.Storage
{
    ...
}
```

Любой код, который вы пишете во вложенном пространстве имен, сможет использовать типы не только из этого пространства имен, но и из содержащих его пространств имен без дополнительной квалификации. Код в листингах 1.7 или 1.8 не требует явной квалификации или директив `using` для использования типов в пространстве имен `MyApp.Storage` или `MyApp`.

Когда вы определяете вложенные пространства имен, условием является создание соответствующей иерархии папок. Как уже было показано, если вы создаете проект с именем `MyApp`, по умолчанию Visual Studio помещает новые классы в пространство имен `MyApp` при добавлении их в проект. Но если вы создадите в проекте новую папку (что можно сделать в `Solution Explorer`) с именем, скажем, `Storage`, то Visual Studio поместит все новые классы, созданные в этой папке, в пространство имен `MyApp.Storage`. Опять же, вам не нужно этого сохранять — Visual Studio просто добавляет объявление пространства имен при создании файла, и вы можете в любое время изменить его. Компилятору не нужно пространство имен, соответствующее вашей иерархии папок. Но поскольку это соглашение поддерживается Visual Studio, жизнь будет гораздо проще, если вы будете ему следовать.

Классы

Внутри объявления пространства имен в моем файле `Program.cs` определен класс. Листинг 1.9 показывает эту часть файла (куда включены и ключевые

слова `public`, которые я добавил ранее). За ключевым словом `class` следует имя, и, конечно, полным именем типа по факту является `HelloWorld.Program`, потому что код находится внутри объявления пространства имен. Как вы можете видеть, в C# используются скобки (`{}`) для разграничения всего и вся — мы уже видели это в случае пространств имен, а здесь вы можете увидеть то же самое в применении к классу, а также к методу, который он содержит.

Листинг 1.9. Класс с методом

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Классы — это механизм C# для определения сущностей, которые сочетают в себе состояние и поведение, что характерно для объектно-ориентированного подхода. Но этот класс не содержит ничего, кроме единственного метода. C# не поддерживает глобальные методы, так что весь код должен принадлежать какому-либо типу. Поэтому этот конкретный класс не очень интересен, ибо его единственная работа — содержать в себе точку входа в программу. Мы увидим еще несколько интересных способов использования классов в главе 3.

Точка входа в программу

По умолчанию компилятор C# будет искать метод с именем `Main` и автоматически использовать его в качестве точки входа. Если вы действительно хотите, вы можете дать команду компилятору использовать другой метод, но большинство программ придерживаются данного соглашения. Независимо от того, назначаете ли вы точку входа из конфигурации или следуете соглашению, метод должен удовлетворять определенным требованиям, все из которых вполне очевидны в листинге 1.9.

Точка входа в *программу* должна быть статическим методом, а это означает, что для вызова метода не нужно создавать экземпляр типа (в данном случае `Program`). Не требуется и возвращать что-либо, что обозначено здесь ключевым словом `void`, хотя если вы хотите, то можете вместо этого вернуть `int`. Это позволит программе возвращать код завершения, который операционная система сообщит, когда программа закончит работу. (Он также может

вернуть либо `Task`, либо `Task<int>`, что позволяет сделать его асинхронным методом, что описано в главе 17.) И метод должен либо вообще не принимать аргументов (что будет обозначаться пустой парой скобок после имени метода), либо, как в листинге 1.9, может принимать один аргумент: массив текстовых строк, содержащих аргументы командной строки.



Некоторые языки семейства С включают имя файла самой программы в качестве первого аргумента на том основании, что оно является частью того, что пользователь вводил в командной строке. C# не следует этому соглашению. Если программа запускается без аргументов, длина массива будет равна 0.

После объявления метода следует тело метода, которое в данном случае содержит код, который почти соответствует тому, что мы хотим. Мы рассмотрели все, что Visual Studio генерировало для нас в этом файле, поэтому осталось только изменить код внутри фигурных скобок, ограничивающих тело метода. Напомню, что тест не пройден, потому что наша программа не отвечает единственному его требованию: вывести определенное сообщение на консоль. Для этого внутри тела метода потребуется одна строка кода, показанная в листинге 1.10. Она почти в точности повторяет ту, что уже есть, просто с дополнительной запятой и строчной буквой `w`.

Листинг 1.10. Отображение сообщения

```
Console.WriteLine("Hello, world!");
```

После этого при повторном запуске тестов `Unit Test Explorer` выводит галочку напротив моего теста и сообщает, что все тесты пройдены. Так что очевидно, что наш код работает. И мы можем самостоятельно это проверить, запустив программу. Это можно сделать с помощью меню `Debug` Visual Studio. Опция `Start Debugging` запускает программу в отладчике. Если вы запустите программу таким образом (что можно сделать и с помощью клавиши `F5`), то откроется окно консоли и вы увидите, что оно отображает наше традиционное сообщение.

Юнит-тесты

Теперь, когда программа работает, вернусь к первому написанному мной коду, а именно тесту, так как он иллюстрирует некоторые особенности C#,

которых нет в основной программе. Если вернуться к листингу 1.1, вы увидите, что он начнется почти так же, как основная программа: там несколько директив `using`, после которых следует объявление пространства имен, на этот раз `HelloWorld.Tests`, что соответствует имени тестового проекта. Но сам класс выглядит иначе. Листинг 1.11 показывает соответствующую часть листинга 1.1.

Листинг 1.11. Тестовый класс с атрибутом

```
[TestClass]
public class WhenProgramRuns
{
```

Непосредственно перед объявлением класса расположен текст `[TestClass]`. Это — *атрибут*. Атрибуты — это примечания, которые можно добавлять к классам, методам и другим объявлениям в коде. Большинство из них ничего не делают сами по себе — компилятор лишь фиксирует тот факт, что атрибут присутствует в скомпилированном выводе, и на этом все. Атрибуты полезны только тогда, когда их ищут, поэтому они, как правило, используются фреймворками. В данном случае я использую среду юнит-теста Microsoft, а она ищет классы, помеченные указанным атрибутом `TestClass`. Она будет игнорировать классы, у которых нет этого примечания. Атрибуты обычно специфичны для конкретной среды, и вы можете определить собственные, как мы увидим в главе 14.

Два метода в классе также снабжены атрибутами. Листинг 1.12 показывает соответствующие выдержки из листинга 1.1. Система выполнения тестов запускает все методы, помеченные `[TestInitialize]`, по разу для каждого теста, содержащегося в классе, и делает это до запуска самого метода теста. И как вы, несомненно, догадались, атрибут `[TestMethod]` сообщает системе выполнения тестов, какие методы представляют собой тесты.

Листинг 1.12. Аннотированные методы

```
[TestInitialize]
public void Initialize()
...
[TestMethod]
public void SaysHelloWorld()
...
```

В листинге 1.1 есть дополнительная особенность: содержимое класса начинается с поля, еще раз показанного в листинге 1.13. Поля содержат данные.

В нашем случае метод `Initialize` сохраняет вывод консоли, который захватывает во время работы программы, в поле `_consoleOutput`, где он доступен для проверки методами тестирования. Это конкретное поле было помечено как `private`, что указывает на то, что оно предназначено для внутреннего использования в пределах класса. Компилятор C# разрешит доступ к этим данным только коду, который содержится в том же классе.

Листинг 1.13. Поле

```
private string _consoleOutput;
```

Итак, мы рассмотрели каждый элемент программы и тестового проекта, который проверяет, что она работает, как задумано.

Итог

Теперь вы имеете представление о базовой структуре программ на C#. Я создал решение, содержащее два проекта, один для тестов и один для самой программы. Листинг был простым, поэтому у каждого проекта только один интересующий нас файл исходного кода. Оба были схожей структуры. Каждый начинался с директивы `using`, указывающих, какие типы использует файл. В объявлении пространства имен указано пространство имен, в котором обитает файл, внутри оно имеет класс, содержащий один или несколько методов или других членов, например полей.

Более подробно мы рассмотрим типы и их элементы в главе 3, но прежде обратимся к главе 2, посвященной коду внутри методов, который и представляет собой то, что мы хотим от нашей программы.

ГЛАВА 2

Основы написания кода на C#

Все языки программирования должны обеспечивать определенные возможности. В первую очередь выразить вычисления и операции в коде. Программы должны иметь возможность принимать решения на основе входных данных. Иногда придется выполнять задачи повторно. Эти фундаментальные особенности являются основой программирования, и в этой главе будет показано, как все это работает в C#.

В зависимости от вашего опыта некоторые материалы этой главы могут показаться очень знакомыми. Считается, что C# принадлежит к «языковому семейству С». Язык С является чрезвычайно влиятельным языком программирования, и многие языки позаимствовали значительную часть его синтаксиса. Есть прямые потомки, такие как C++ и Objective-C. Есть также языки с меньшей степенью родства, включая Java, JavaScript и собственно C#. Они не совместимы с С, но все еще содержат многие аспекты его синтаксиса. Если вы знакомы с любым из этих языков, то узнаете многие языковые особенности, которые мы собираемся изучить.

Мы уже совершили обзор основных элементов программы в главе 1. В этой главе мы сосредоточимся только на коде внутри методов. Как вы видели, C# требует определенной структуры кода: он состоит из операторов, расположенных внутри метода, который принадлежит к типу, обычно находящемуся внутри пространства имен; все это внутри файла, являющегося частью проекта, как правило, внутри решения. Для наглядности большинство примеров в этой главе будут показывать интересующий код изолированно, как в листинге 2.1.

Листинг 2.1. Код и ничего, кроме кода

```
Console.WriteLine("Hello, world!");
```

Если не оговорено иное, такой отрывок является сокращенной записью кода в контексте соответствующей программы. Таким образом, листинг 2.1 является сокращением для листинга 2.2.

Листинг 2.2. Весь код

```
using System;

namespace Hello
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello, world!");
        }
    }
}
```

Хотя в этом разделе я расскажу о фундаментальных элементах языка, эта книга предназначена для тех, кто уже знаком хотя бы с одним языком программирования. Поэтому тривиальные возможности я опишу кратко и постараюсь углубиться в те аспекты, которые характерны для C#.

Локальные переменные

В неистребимом примере `Hello, world!` отсутствует жизненно важный элемент: на самом деле он не обрабатывает информацию. Полезные программы обычно извлекают, обрабатывают и производят информацию, поэтому способность определять и идентифицировать информацию является одной из наиболее важных особенностей языка. Как и большинство других языков, C# позволяет вам определять локальные переменные, которые являются именованными элементами внутри метода, каждый из которых содержит частицу информации.



В спецификации C# термин «переменная» может относиться как к локальным переменным, так и к полям в объектах и элементам массива. Этот раздел полностью посвящен локальным переменным, но постоянно натыкаться на префикс `local` довольно утомительно. Так что с этого момента в текущем разделе переменная означает локальную переменную.

C# – это язык со статической типизацией, т. е. любой элемент кода, который представляет или производит информацию (например, переменная или выражение), имеет тип данных, определенный во время компиляции. Это

отличается от языков с динамической типизацией, таких как JavaScript, в которых типы определяются во время выполнения.

Самый простой способ увидеть статическую типизацию C# в действии — взглянуть на простые объявления переменных, как в листинге 2.3. Каждый из них начинается с типа данных — первые две переменные имеют тип `string`, за которыми следуют две переменные типа `int`¹. Эти типы представляют собой текстовые строки и 32-разрядные целые числа со знаком соответственно.

Листинг 2.3. Объявления переменных

```
string part1 = "the ultimate question";
string part2 = "of something";
int theAnswer = 42;
int andAnotherThing;
```

За типом данных немедленно следует имя переменной. Оно должно начинаться либо с буквы, либо со знака подчеркивания, за которым может следовать любая комбинация букв, десятичных цифр и подчеркиваний. (По крайней мере, это те варианты, которые диктует ASCII. Язык C# поддерживает Юникод, поэтому, если вы сохраните свой файл в формате UTF-8 или UTF-16, все, что находится после первого символа в идентификаторе, может быть любым из символов, описанных в приложении к спецификации Юникод «Идентификатор и синтаксис шаблона». Это включает в себя различные акценты, диакритические знаки и многочисленные несколько непонятные знаки препинания. Нельзя задействовать только символы, предназначенные для использования внутри слов, например символы, которые Юникод идентифицирует как предназначенные для разделения слов.) Эти же правила определяют, что представляет собой допустимый идентификатор для любой определенной пользователем сущности в C#, такой как класс или метод.

Листинг 2.3 показывает, что есть несколько форм объявлений переменных. Первые три переменные включают блок инициализации, который устанавливает начальное значение переменной, но, как показывает последняя переменная, это необязательно. Это потому, что вы можете назначать новые значения переменным в любое время. Листинг 2.4 продолжает листинг 2.3

¹ C# предлагает динамическую типизацию в качестве опции со своим ключевым словом `dynamic`, но делает несколько необычный шаг в сторону статической типизации: динамические переменные имеют статический тип `dynamic`.

и показывает, что можно присвоить новое значение переменной независимо от того, имела ли она начальное значение.

Листинг 2.4. Присвоение значений ранее объявленным переменным

```
part2 = " of life, the universe, and everything";  
andAnotherThing = 123;
```

Поскольку переменные имеют статический тип, компилятор будет отклонять все попытки назначить неправильный тип данных. Так что, если бы мы дополнили листинг 2.3 кодом листинга 2.5, компилятор начал бы жаловаться. Ему известно, что переменная с именем `Answer` имеет тип `int`, который является числовым типом, поэтому он сообщит об ошибке, если мы попытаемся присвоить ей текстовую строку.

Листинг 2.5. Ошибка: неправильный тип

```
theAnswer = "The compiler will reject this";
```

Вам бы разрешили это сделать в динамических языках, таких как JavaScript, потому что в них переменная не имеет собственного типа. Все, что имеет значение, — это значимый тип, которое она содержит, а он может измениться по мере выполнения кода. Нечто подобное можно сделать и в C#, объявив переменную с типом `dynamic` или `object` (что я опишу позже в подразделах «Тип `dynamic`» на с. 112 и «Тип `object`» на с. 114). Тем не менее наиболее распространенной практикой в C# является использование переменных более определенного типа.



Благодаря наследованию статический тип не всегда отражает полную картину. Я буду обсуждать это в главе 6, но пока достаточно знать, что некоторые типы можно расширить посредством наследования, и если переменная использует такой тип, то можно сослаться на какой-то объект, тип которого унаследован от статического типа переменной. Подобную же гибкость обеспечивает использование интерфейсов, описанных в главе 3. Однако статический тип неизменно определяет, какие операции разрешено выполнять с переменной. Если вы хотите использовать дополнительные члены, специфичные для некоторого производного типа, вам не позволено делать это через переменную базового типа.

Вам не нужно явно указывать тип переменной. Вы можете позволить компилятору решить это за вас, используя ключевое слово `var` вместо типа данных.

Листинг 2.6 повторяет первые три объявления переменных из листинга 2.3, но с использованием `var` вместо явных типов данных.

Листинг 2.6. Неявные типы переменных с ключевым словом `var`

```
var part1 = "the ultimate question";
var part2 = "of something";
var theAnswer = 40 + 2;
```

Этот код способен смутить людей, немного знакомых с JavaScript, потому что в нем также есть ключевое слово `var`, которое вы можете использовать подобным же образом. Но в C# `var` работает не так, как в JavaScript: здесь переменные остаются статически типизированными. Изменения заключаются в том, что мы не сказали, что это за тип, позволив компилятору сделать это за нас. Он смотрит на блоки инициализации и видит, что первые две переменные — это строки, а третья — целое число. (Вот почему я пропустил четвертую переменную из листинга 2.3, `andAnotherThing`. У нее нет блока инициализации, поэтому у компилятора нет возможности определить ее тип. Если вы попытаетесь использовать ключевое слово `var` без инициализации, то получите ошибку компилятора.)

Можно проверить, что переменные, объявленные с помощью `var`, статически типизированы, попытавшись присвоить им значение другого типа. Мы могли бы повторить проделанное в листинге 2.5, но на этот раз с переменной типа `var`. Листинг 2.7 делает это, и компилятор выдает точно такую же ошибку, так как мы пытаемся присвоить текстовую строку переменной несовместимого типа. Эта переменная, `theAnswer`, имеет тип `int`, хотя мы и не указали это явно.

Листинг 2.7. Ошибка: неверный тип (еще раз)

```
var theAnswer = 42;
theAnswer = "The compiler will reject this";
```

Мнения относительно того, как и когда использовать ключевое слово `var`, разделились. См. врезку «Var или не var?».

Последнее, что стоит знать об объявлениях, это то, что вы можете объявить и, возможно, инициализировать сразу несколько переменных в одной строке. Если вам нужно несколько переменных одного типа, то эта возможность уменьшит беспорядок в вашем коде. Листинг 2.8 объявляет три переменные одного типа в одном объявлении.

VAR ИЛИ НЕ VAR?

Переменная, объявленная с помощью `var`, ведет себя точно так же, как и эквивалентное ей объявление с явным типом, что вызывает вопрос: какой вариант использовать? В некотором смысле это не имеет значения, потому что переменные эквивалентны. Однако, если вы хотите, чтобы ваш код был последовательным, нужно выбрать один стиль и придерживаться его. Ведутся споры, какой стиль является «лучшим».

Некоторые разработчики считают дополнительный текст, необходимый для указания явных типов переменных, непродуктивными «церемониями», предпочитая более короткое ключевое слово `var`. Аргумент звучит так: позвольте компилятору определить тип для вас, вместо того чтобы делать работу самим. Кроме того, это визуально упорядочивает код.

Я придерживаюсь другого взгляда, потому что трачу больше времени на чтение кода, чем на его написание, ставя во главу угла отладку, обзор кода, рефакторинг и улучшение. Все, что облегчает эти действия, стоит небольшого времени, которое требуется для явного указания типов. Код, который везде использует `var`, замедляет вашу работу, потому что нужно понять, что это на самом деле за тип. Хотя `var` сэкономил вам определенные усилия в процессе написания кода, этот выигрыш быстро исчезает из-за дополнительных размышлений, которые требуются каждый раз, когда вы возвращаетесь к этому коду. Так что если вы не тот разработчик, который лишь единожды пишет новый код, заставляя затем других в нем разбираться, то единственное преимущество философии «`var` везде» — это потенциальная точность кода.

Вы даже можете использовать явные типы и при этом заставить компилятор выполнять свою работу: в Visual Studio можно написать экономное `var`, затем нажать `Ctrl +`, что откроет меню `Quick Actions`, где будут перечислены предложения замены на явный тип. (Для определения типа переменной Visual Studio использует API компилятора C#.)

Тем не менее есть некоторые ситуации, в которых я буду использовать `var`. Одна из них состоит в том, чтобы не писать имя типа дважды, как в этом примере:

```
List<int> numbers = new List<int>();
```

Для простоты мы можем отбросить первый `List<int>`, потому что имя все еще здесь, в блоке инициализации. Есть похожие примеры, связанные с преобразованиями типов и общими методами. Пока имя типа явно указывается в объявлении переменной, вполне можно использовать `var`, чтобы избежать написания типа дважды.

Я также использую `var` там, где это действительно нужно. Как мы увидим в следующих главах, C# поддерживает анонимные типы, и, как следует из названия, невозможно указать имя такого типа. В этих ситуациях вы можете быть вынуждены использовать `var`. (На самом деле ключевое слово `var` было введено в C# только с добавлением анонимных типов.)

Листинг 2.8. Несколько переменных в одном объявлении

```
double a = 1, b = 2.5, c = -3;
```

Независимо от объявления переменная содержит некоторую информацию определенного типа, и компилятор не позволяет нам помещать в эту переменную данные несовместимого типа. Переменные полезны только потому, что мы можем обратиться к ним позже. Листинг 2.9 начинается с объявлений переменных, которые мы уже видели в предыдущих примерах, затем использует значения этих переменных для инициализации некоторых других переменных, после чего отображает результаты.

Листинг 2.9. Использование переменных

```
string part1 = "the ultimate question";
string part2 = "of something";
int theAnswer = 42;

part2 = "of life, the universe, and everything";

string questionText = "What is the answer to " + part1 + ",
                      " + part2 + "?";
string answerText = "The answer to " + part1 + ", " +
                    part2 + ", is: " + theAnswer;

Console.WriteLine(questionText);
Console.WriteLine(answerText);
```

Кстати, этот код опирается на тот факт, что C# определяет несколько значений для оператора +, когда используется со строками. Во-первых, когда вы «складываете» две строки вместе, он объединяет их. Во-вторых, когда вы «добавляете» что-то, кроме строк, в конец строки (как это делает инициализатор для answerText — он добавляет ответ, который является числом), C# генерирует код, который перед добавлением преобразует значение в строку. Таким образом, листинг 2.9 выводит:

```
What is the answer to the ultimate question, of life, the universe,
and everything?
The answer to the ultimate question, of life, the universe, and
everything, is: 42
```

Когда вы используете переменную, ее значением является то, что вы в последний раз ей присвоили. Если вы попытаетесь использовать переменную до присвоения значения, как в листинге 2.10, компилятор C# сообщит об ошибке.

Листинг 2.10. Ошибка: использование переменной без присвоенного значения

```
int willNotWork;  
Console.WriteLine(willNotWork);
```

Попытка компиляции покажет следующую ошибку во второй строке:

```
error CS0165: Use of unassigned local variable 'willNotWork'
```



В этой книге текст длиной более 80 символов переносится на несколько строк, чтобы соответствовать размеру страницы. Когда вы запустите эти примеры, результаты могут выглядеть иначе, если окна вашей консоли настроены на другую ширину.

Компилятор использует слегка пессимистичную систему (которую он называет правилами *определенного присваивания*) для определения, содержит ли переменная значение. Невозможно создать алгоритм, который может точно определять подобные вещи в любой возможной ситуации. Так как компилятор вынужден подстраховываться, в некоторых ситуациях переменная уже будет иметь значение на момент запуска кода, который вызывает ошибку, и все же компилятор будет жаловаться². Решение состоит в добавлении блока инициализации, чтобы переменная всегда содержала хоть что-то, скажем `0` для числовых значений и `false` для булевых переменных. В главе 3 я представлю ссылочные типы, и, как следует из названия, переменная такого типа может содержать ссылку на экземпляр типа. Если вам нужно инициализировать такую переменную до того, как у вас появится что-то, на что она может ссылаться, вы можете использовать специальное ключевое слово `null`, обозначающее ссылку на ничто.

Правила определенного присваивания определяют те части вашего кода, в которых, как считает компилятор, переменная содержит допустимое значение, и поэтому вам позволено читать из нее. Запись в переменную менее ограничена, но, как и следовало ожидать, любая переменная доступна только из определенных частей кода. Давайте взглянем на правила, которые всем этим управляют.

² См. фундаментальную работу Алана Тьюринга по вычислениям. Книга *The Annotated Turing* Чарльза Петцольда станет отличным путеводителем по этой работе.

Область видимости

Область видимости переменной — это диапазон кода, в котором вы можете обращаться к этой переменной по ее имени. Переменные — не единственное, что обладает областью видимости. Методы, свойства, типы и на самом деле все, что имеет имя, имеет и область видимости. Такое утверждение требует расширения определения области видимости: это части вашего кода, где вы можете ссылаться на объект по его имени, не требуя дополнительной квалификации. Когда я пишу `Console.WriteLine`, то обращаюсь к методу по его имени (`WriteLine`), но мне нужно квалифицировать его именем класса (`Console`), потому что метод не находится в области видимости. Но в случае с локальной переменной область действия является абсолютной: либо она доступна без квалификации, либо она вообще недоступна.

Вообще говоря, область действия переменной начинается с ее объявления и заканчивается в конце содержащего ее блока. (Конструкции цикла, к которым мы обратимся позже, приводят к паре исключений из этого правила.) Блок — это область кода, ограниченная парой скобок (`{}`). Тело метода является блоком, поэтому переменная, определенная в одном методе, не видна в другом, потому что он за пределами области видимости. Если вы попытаетесь скомпилировать листинг 2.11, вы получите сообщение об ошибке: `The name 'thisWillNotWork' does not exist in the current context.`

Листинг 2.11. Ошибка: вне области видимости

```
static void SomeMethod()
{
    int thisWillNotWork = 42;
}

static void AnUncompilableMethod()
{
    Console.WriteLine(thisWillNotWork);
}
```

Методы часто содержат вложенные блоки, особенно когда вы работаете с конструкциями цикла и управления потоком, которые мы рассмотрим позже в этой главе. На момент начала вложенного блока все, что находится в области видимости во внешнем блоке, продолжает оставаться в области видимости внутри этого вложенного блока. В листинге 2.12 объявляется переменная с именем `someValue`, за чьим следом следует вложенный блок как часть

оператора `if`. Код внутри этого блока может получить доступ к переменной, объявленной в содержащем блоке.

Листинг 2.12. Объявленная вне блока переменная используется внутри блока

```
int someValue = GetValue();
if (someValue > 100)
{
    Console.WriteLine(someValue);
}
```

Обратное неверно. Если вы объявляете переменную во вложенном блоке, ее область действия не выходит за пределы этого блока. Таким образом, код из листинга 2.13 не удастся скомпилировать, потому что переменная `willNotWork` находится в области видимости только вложенного блока. Последняя строка кода выдаст ошибку компилятора из-за попытки использовать эту переменную вне этого блока.

Листинг 2.13. Ошибка: попытка использовать переменную вне области видимости

```
int someValue = GetValue();
if (someValue > 100)
{
    int willNotWork = someValue - 100;
}
Console.WriteLine(willNotWork);
```

Все это может показаться довольно простым, но становится гораздо сложнее, когда дело доходит до возможных конфликтов имен. Здесь C# иногда способен захватить врасплох.

Неопределенность имени переменной

Рассмотрим код в листинге 2.14. Он объявляет переменную с именем `anotherValue` во вложенном блоке. Как вы знаете, область видимости этой переменной распространяется только до конца вложенного блока. После окончания этого блока мы попытаемся объявить другую переменную с тем же именем.

Листинг 2.14. Ошибка: неожиданный конфликт имен

```
int someValue = GetValue();
if (someValue > 100)
{
    int anotherValue = someValue - 100; // Compiler error
```

```
        Console.WriteLine(anotherValue);
    }

int anotherValue = 123;
```

Это приводит к ошибке компилятора в первой из строк, объявляющих `anotherValue`:

```
error CS0136: A local or parameter named 'anotherValue' cannot be
declared in this scope because that name is used in an enclosing
local scope to define a local or parameter
```

Это кажется странным. Объявление, в последней строке предположительно конфликтующее с более ранним, лежит вне его области видимости, потому что мы находимся за пределами вложенного блока. Более того, второе объявление не входит в область видимости этого вложенного блока, потому что объявление идет после блока. Области не перекрываются, но, несмотря на это, мы нарушили правило C# относительно конфликтов имен. Чтобы понять, почему этот код приводит к ошибке, сначала стоит взглянуть на более прозаический пример.

C# пытается предотвратить неоднозначность, запрещая код, в котором одно имя может относиться к нескольким сущностям. Листинг 2.15 показывает тип проблемы, которую стремится предотвратить компилятор. Здесь у нас есть переменная с именем `errorCount`, и код начинает изменять ее по мере работы. На полпути во вложенном блоке появляется еще одна переменная, также названная `errorCount`³. Вполне можно представить язык, который позволил бы такое, — в нем могло бы быть правило, которое гласит, что когда в область действия входят несколько элементов с одним и тем же именем, будет выбран тот, объявление которого произошло последним.

Листинг 2.15. Ошибка: скрытие переменной

```
int errorCount = 0;
if (problem1)
{
    errorCount += 1;

    if (problem2)
```

³ Если вы новичок в языках семейства C, оператор `+=` может показаться незнакомым. Это составной оператор присваивания, описанный далее в этой главе. Я использую его, чтобы увеличить `errorCount` на единицу.

```
{  
    errorCount += 1;  
}  
  
// Представьте себе, что в настоящей программе здесь большой  
// кусок кода перед следующими строками.  
  
int errorCount = GetErrors(); // Ошибка компиляции  
if (problem3)  
{  
    errorCount += 1;  
}  
}
```

C# не позволяет этого, потому что код, в котором это допустимо, легко введет в заблуждение. Это намеренно короткий и наглядный пример, позволяющий легко увидеть повторяющиеся имена. Но если бы код был немного длиннее, было бы очень легко пропустить объявление вложенных переменных. Тогда мы можем не понять, что в конце метода `errorCount` ссылается уже на что-то другое. Во избежание недоразумений C# попросту это запрещает.

Но в чем же ошибка в листинге 2.14? Области видимости двух переменных не пересекаются. Выходит, что в основе правила, запрещающего компиляцию листинга 2.15, лежат не области видимости. Оно основано на немного другой концепции, называемой пространством объявлений. Пространство объявления — это область кода, в которой одно имя не должно ссылаться на два разных объекта. Каждый метод устанавливает пространство объявления для переменных. Вложенные блоки также устанавливают пространства объявлений, и во вложенном пространстве объявлений недопустимо объявлять переменную с тем же именем, что и в родительском пространстве объявлений. Это и есть то правило, которое мы здесь нарушили, — внешнее пространство объявлений в листинге 2.15 содержит переменную `errorCount`, а пространство объявлений вложенного блока пытается ввести другую переменную с тем же именем.

Если все это кажется необоснованным, будет полезным знать, почему есть целый отдельный набор правил для коллизий имен, вместо того чтобы основываться на областях видимости. Цель правила пространства объявлений заключается в том, что в большинстве случаев не должно иметь значения, куда именно вы помещаете объявление. Если бы вам нужно было переставить все объявления переменных в блоке в начало этого блока, — а в некоторых компаниях есть стандарты написания кода, которые предписывают

такой тип компоновки, — идея этих правил заключается в том, что это не должно изменить смысл кода. Очевидно, что это было бы невозможно, если бы листинг 2.15 был дозволенным. Это же объясняет, почему недопустимым является и листинг 2.14. Хотя области и не перекрываются, они бы смешились, если бы вы переместили все объявления переменных в верхнюю часть содержащих их блоков.

Экземпляры локальной переменной

Переменные являются элементами исходного кода, поэтому каждая конкретная переменная идентифицируется отдельно: она объявляется ровно в одном месте исходного кода и выходит из области видимости в одном, четко определенном месте. Но это не означает, что ей соответствует единственное место хранения в памяти. Один и тот же метод может быть вызван одновременно через рекурсию, многопоточность или асинхронное выполнение.

Каждый раз при запуске метод получает отдельный набор хранилищ для значений локальных переменных. Это позволяет нескольким потокам без проблем одновременно выполнять один и тот же метод, поскольку каждый из них имеет свой собственный набор локальных переменных. Аналогично в рекурсивном коде каждый вложенный вызов получает свой собственный набор локальных объектов, которые не будут мешать ни одному из предшествующих. То же самое касается нескольких одновременных вызовов метода. Если подходить к вопросу максимально строго, то каждое выполнение конкретной области видимости получает собственный набор переменных. Это различие имеет значение, когда используются анонимные функции (см. главу 9). В целях оптимизации C#, когда это возможно, использует места хранения повторно, поэтому для выполнения каждой области видимости память выделяется лишь тогда, когда это действительно необходимо (например, она не будет при каждой итерации выделяться для переменных, объявленных в теле цикла, если только вы не создадите ситуацию, когда не остается иного выбора), но выглядит все так, как если бы каждый раз выделялось новое пространство.

Имейте в виду, что компилятор C# не дает никаких особых гарантий относительно того, где именно находятся переменные (кроме исключительных случаев, как мы увидим в главе 18). Они вполне могут быть в стеке, а могут и не быть. Когда в последующих главах мы рассмотрим анонимные функции, вы увидите, что переменные иногда должны переживать метод, который

их объявляет, потому что они остаются в области действия для вложенных методов, которые будут запускаться как обратные вызовы после возврата из содержащего метода.

Кстати, прежде чем мы продолжим, учитывайте, что так же, как переменные не являются единственными объектами, имеющими область видимости, они также не являются единственными объектами, к которым применяются правила пространства объявлений. Позже мы рассмотрим другие элементы языка, в том числе классы, методы и свойства, к которым также применяются правила области видимости и уникальности имен.

Инструкции и выражения

Переменные дают нам возможность разместить информацию, с которой работает наш код, но, чтобы что-то сделать с этими переменными, нужно написать какой-то код. Это будет означать написание инструкций и выражений.

Инструкции

Когда мы пишем метод C#, то пишем последовательность инструкций, или операторов. Говоря простыми словами, операторы в методе описывают действия, которые должны выполняться. Каждая строка в листинге 2.16 является инструкцией. Может возникнуть соблазн думать об инструкции как об указании для выполнения чего-то одного (например, инициализации переменной или вызова метода). Можно избрать более лексикографический подход, когда все, что заканчивается точкой с запятой, — это инструкция, или оператор. (И кстати, значение здесь имеют именно точки с запятой, а не разрывы строк. Мы могли бы написать все это как одну длинную строку кода, и это было бы совершенно эквивалентным кодом.) Однако оба описания упрощенные, даже если они и верны для этого конкретного примера.

Листинг 2.16. Некоторые инструкции

```
int a = 19;
int b = 23;
int c;
c = a + b;
Console.WriteLine(c);
```

C# распознает множество разных видов инструкций (операторов). Первые три строки листинга 2.16 – это операторы объявления, которые объявляют и, необязательно, инициализируют переменную. Четвертая и пятая строки – это операторы выражений. Но некоторые операторы имеют более сложную структуру, чем показано в этом примере.

Когда вы пишете цикл, это оператор цикла. Когда вы используете механизмы `if` или `switch`, описанные далее в этой главе, для выбора между различными возможными действиями, это условные операторы. Фактически спецификация C# различает 13 категорий операторов. Большинство из них в целом можно описать как указание того, что код должен делать дальше, или, для таких функций, как циклы или условные операторы, как описание того, как ему решить, что делать дальше. Операторы второго типа обычно содержат один или несколько встроенных операторов, описывающих действие, которое нужно выполнить в цикле, или действие, которое нужно выполнить при выполнении условия оператора `if`.

Хотя есть один особый случай. Блок – это своего рода оператор. Это делает такие операторы, как циклы, более полезными, чем они могли бы быть, потому что цикл повторяет единственный встроенный оператор. Этот оператор может быть блоком, и поскольку сам блок является последовательностью операторов (ограниченных фигурными скобками), это позволяет циклам содержать более одного оператора.

Это показывает, почему две упрощенные точки зрения, изложенные ранее, – «операторы – это действия» и «операторы – это все, что заканчивается точкой с запятой», – ошибочны. Сравните листинг 2.16 с листингом 2.17. Оба делают одно и то же, потому что различные действия, которые мы в них отразили, остаются одинаковыми, плюс оба они содержат по пять точек с запятой. Однако листинг 2.17 содержит один дополнительный оператор. Первые два оператора совпадают, но за ними следует третий оператор, блок, который включает последние три оператора из листинга 2.16. Дополнительный оператор не заканчивается точкой с запятой и не выполняет никаких действий. В этом конкретном примере это бессмысленно, но иногда полезно использовать такой вложенный блок, чтобы избежать ошибок, связанных с неоднозначностью имен. Таким образом, операторы могут быть структурными, а не выполнять какое-либо действие во время выполнения.

Хотя ваш код будет содержать смесь типов операторов, он неизбежно будет содержать по крайней мере несколько операторов выражений. Это просто

операторы, которые состоят из допустимого выражения, за которым следует точка с запятой. Что же такое допустимое выражение? Что такое выражение, если уж на то пошло? Лучше ответить на этот второй вопрос, прежде чем вернуться к тому, что представляет собой допустимое для оператора выражение.

Листинг 2.17. Блок

```
int a = 19;
int b = 23;
{
    int c;
    c = a + b;
    Console.WriteLine(c);
}
```

Выражения

Официальное определение выражения в C# довольно сухое: «последовательность операторов и операндов». Следует признать, что спецификации языка, как правило, такими и бывают, но в дополнение к такого рода формальной прозе спецификация C# содержит кое-какие очень понятные неформальные объяснения более формально выраженных идей. (Например, оператор описывается как средство, с помощью которого «выражаются действия программы», после чего следует менее доступное описание, но с использованием более технически точного языка.)

Цитата в начале этого абзаца взята из формального определения выражения, поэтому можно было надеяться, что неформальное объяснение из введения окажется более полезным. Но нам не повезло: в нем говорится, что выражения «составлены из операндов и операторов». Это, безусловно, менее точно, чем другое определение, но понять его не легче. Проблема в том, что есть несколько видов выражений, которые выполняют разные задачи, поэтому не существует единого общего неформального описания.

Возникает соблазн описать выражение как некий код, результатом которого является какое-то значение. Хоть это и не относится ко всем выражениям, большинство выражений, которые вы напишете, будут соответствовать этому описанию. Поэтому сейчас я сосредоточусь на нем, а позже расскажу об исключениях.

Простейшими выражениями являются *литералы*, в которые мы просто записываем желаемое значение, например "Hello, world!" или 42. Вы также

можете использовать в качестве выражения имя переменной. Выражения могут включать операторы, которые описывают вычисления или другие необходимые расчеты. Операторы имеют фиксированное количество входных данных, которые называются операндами. Некоторые принимают лишь один operand. Например, вы можете сделать число отрицательным, поставив перед ним знак минус. Некоторые принимают два: оператор + позволяет вам сформировать выражение, которое складывает результаты двух operandов по обе стороны от символа +.



Некоторые символы играют различные роли в зависимости от контекста. Знак минус используется не только для отрицания. Он действует как оператор вычитания с двумя operandами, если появляется между двумя выражениями.

В общем, operandы также являются выражениями. Итак, когда мы пишем `2 + 2`, то мы пишем выражение, которое содержит еще два выражения — пару литералов '2' по обе стороны от символа +. Это означает, что мы можем писать произвольно сложные выражения, вкладывая выражения в выражения. Листинг 2.18 использует это для вычисления квадратичной формулы (стандартная методика решения квадратных уравнений).

Листинг 2.18. Выражения в выражениях

```
double a = 1, b = 2.5, c = -3;  
double x = (-b + Math.Sqrt(b * b - 4 * a * c)) / (2 * a);  
Console.WriteLine(x);
```

Взгляните на оператор объявления во второй строке. Общая структура его блока инициализации, который является выражением, представляет собой операцию деления. Но два operandы этого оператора деления также являются выражениями. Его левый operand является *выражением в скобках*, которое говорит компилятору, что я хочу, чтобы все выражение `(-b + Math.Sqrt (b * b - 4 * a * c))` стало первым operandом деления. Это подвыражение содержит сложение, левый operand которого является выражением отрицания, чей единственный operand является переменной `b`. Правая часть сложения берет квадратный корень другого, более сложного выражения. А правый operand деления — это другое выражение в скобках, содержащее умножение. На рис. 2.1 показана полная структура выражения.

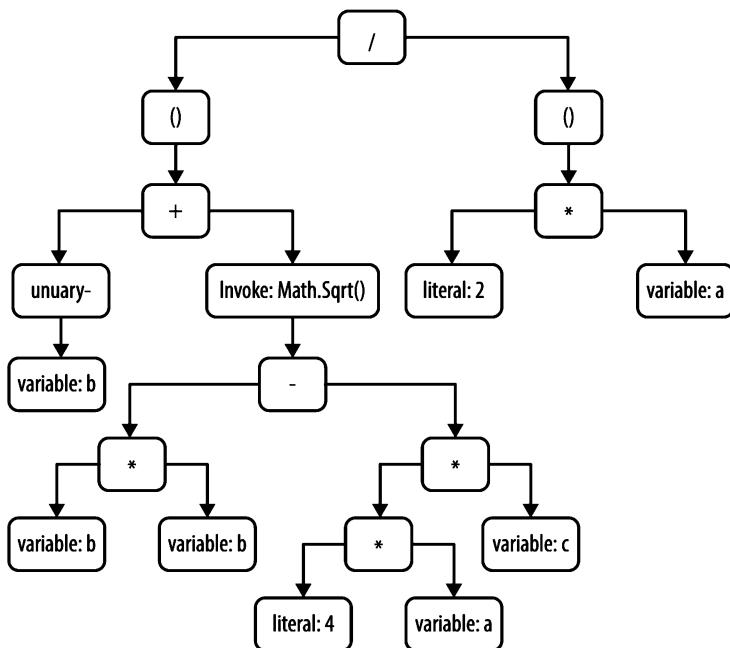


Рис. 2.1. Структура выражения

Важной деталью этого последнего примера является то, что вызовы методов являются своего рода выражениями. Метод `Math.Sqrt`, используемый в листинге 2.18, является функцией библиотеки классов .NET, которая вычисляет квадратный корень из ее входных данных и возвращает результат. Возможно, еще более удивительным является то, что вызовы методов, которые не возвращают значение, например `Console.WriteLine`, технически также являются выражениями. Есть ряд других конструкций, которые не возвращают значений, но все еще считаются выражениями, включая ссылку на тип (например, `Console` в `Console.WriteLine`) или на пространство имен. В силу того что они являются выражениями, эти виды конструкций используют набор общих правил (например, область видимости, определение того, что относится к имени, и т. д.). Однако все не производящие значений выражения могут использоваться только в определенных конкретных обстоятельствах. (Например, вы не можете использовать один из них как операнд в другом выражении.) Поэтому, хотя технически неправильно определять выражения как фрагменты кода, которые производят значения, именно такие выражения мы используем, описывая, какие вычисления должен выполнять наш код.

Теперь можно вернуться к вопросу: что допустимо поместить в оператор выражения? Грубо говоря, выражение должно что-то делать; оно не может просто вычислить значение. Таким образом, хотя `2 + 2` является допустимым выражением, вы получите ошибку, если попытаетесь превратить его в оператор выражения, поставив в конце точку с запятой. Это выражение что-то вычисляет, но ничего не делает с результатом. Если быть более точным, в качестве операторов можно использовать следующие выражения: вызов метода, присваивание, приращение, уменьшение и создание нового объекта. Мы поговорим об увеличении и уменьшении чуть дальше в этой главе, а к объектам обратимся в последующих главах, что оставляет нам только вызовы и присваивание.

Таким образом, вызов метода может быть оператором выражения. Он может включать вложенные выражения других видов, но все это в целом должно быть вызовом метода. В листинге 2.19 показано несколько допустимых вариантов. Обратите внимание, что компилятор C# не проверяет, действительно ли вызов метода приводит к какому-либо продолжительному действию — функция `Math.Sqrt` является чистой функцией в том смысле, что она не делает ничего, кроме возвращения значения, полностью определенного входными данными. Так что вызвать ее и потом ничего не сделать с результатом означает не сделать вообще ничего. В этом не больше действия, чем в выражении `2 + 2`. Но что касается компилятора C#, в качестве оператора выражения допускается любой вызов метода.

Листинг 2.19. Выражения вызова метода как операторы

```
Console.WriteLine("Hello, world!");
Console.WriteLine(12 + 30);
Console.ReadKey();
Math.Sqrt(4);
```

Кажется непоследовательным, что C# запрещает использовать выражение сложения в качестве оператора, хотя использовать `Math.Sqrt` можно. Оба выполняют вычисления, которые дают результат, поэтому нет смысла в таком использовании любого из них. Разве не более логичным для C# было бы допускать только те вызовы методов, которые не возвращают ничего, что можно использовать как операторы выражения? Это исключило бы заключительную строку листинга 2.19, что кажется неплохой идеей, потому что этот код не делает ничего полезного. Это также согласуется с тем фактом, что `2 + 2` также не может быть оператором выражения. К сожалению, иногда требуется игнорировать возвращаемое значение. Листинг 2.19 вызывает

метод `Console.ReadKey()`, который ожидает нажатия клавиши и возвращает значение, указывающее, какая именно клавиша была нажата. Если поведение моей программы зависит от того, какую конкретную клавишу нажал пользователь, мне, конечно, нужно будет проверить возвращаемое значение метода. Но если я просто хочу дождаться какой-либо клавиши вообще, то можно смело игнорировать возвращаемое значение. Если бы C# не позволял использовать методы с возвращаемыми значениями в качестве операторов выражений, мне бы не удалось этого сделать. Компилятор не способен отличить методы, из которых в силу отсутствия побочных эффектов получаются бессмысленные операторы (например, `Math.Sqrt`), и методы, которые могут быть хорошими кандидатами на эту роль (например, `Console.ReadKey`), поэтому допускает любой метод.

Чтобы быть допустимым оператором выражения, недостаточно просто содержать вызов метода. В листинге 2.20 показан ряд выражений, которые вызывают методы, а затем используют их как часть выражений сложения. Хотя это допустимые выражения, они не являются допустимыми операторами выражений и поэтому вызывают ошибки компилятора. Что имеет значение, так это внешнее выражение. В обеих строках это выражение сложения, поэтому они недопустимы.

Листинг 2.20. Ошибки: некоторые выражения, которые не могут быть операторами

```
Console.ReadKey().KeyChar + "!";
Math.Sqrt(4) + 1;
```

Ранее я говорил, что одним из видов выражений, которые разрешено использовать в качестве операторов, является присваивание. То, что присваивания должны быть выражениями, не очевидно, но это так, и они действительно дают значение: результатом выражения присваивания служит значение, присваиваемое переменной. Это означает, что вполне допустимо писать код, подобный тому, что мы видим в листинге 2.21. Во второй строке здесь выражение присваивания используется в качестве аргумента для вызова метода, который показывает значение этого выражения. Первые два вызова `WriteLine` оба отображают 123.

Вторая часть этого примера присваивает одно значение двум переменным за один шаг, используя тот факт, что присваивания являются выражениями — переменной `x` присваивается значение выражения `y = 0` (равное 0).

Это показывает, что вычислением выражения можно добиться большего, чем просто получить значение. У некоторых выражений есть побочные

эффекты. Мы только что увидели, что присваивание является выражением, и, конечно, оно меняет то, что находится в переменной. Вызовы методов также являются выражениями, и хотя вы можете писать чистые функции, которые ничего не делают, кроме вычислений на основе входных данных, например `Math.Sqrt`, многие методы делают что-то более долгоиграющее, вроде записи данных на экран, обновления базы данных или запуска ракеты. Это означает, что нам стоило бы позаботиться о порядке вычисления операндов выражения.

Листинг 2.21. Присваивания — это выражения

```
int number;  
Console.WriteLine(number = 123);  
Console.WriteLine(number);  
  
int x, y;  
x = y = 0;  
Console.WriteLine(x);  
Console.WriteLine(y);
```

Структура выражения накладывает некоторые ограничения на порядок, в котором операторы выполняют свою работу. Например, для упорядочивания я могу использовать скобки. Выражение $10 + (8/2)$ возвращает 14, а выражение $(10 + 8)/2$ возвращает 9, хотя оба имеют одинаковые лiteralные операнды и арифметические операторы. Скобки здесь определяют, выполняется ли деление до или после вычитания⁴.

Однако хотя структура выражения накладывает некоторые ограничения на порядок, она же и оставляет некоторую свободу: хотя оба операнда сложения должны быть вычислены до самого сложения, оператору сложения не важно, какой operand мы вычисляем первым. Но если operandы являются выражениями с побочными эффектами, порядок может иметь значение. Для подобных простых выражений это не имеет значения, потому что я использовал литералы, так что мы не можем точно сказать, когда именно они вычисляются. Но как быть с выражением, в котором operandы вызывают какой-то метод? Листинг 2.22 содержит такой код.

⁴ В отсутствие скобок C# применяет правило первоочередности, которые определяют порядок, в котором вычисляются операторы. Для получения полной (и не очень интересной) информации обратитесь к документации. В этом же примере деление имеет более высокий приоритет, чем сложение, так что без каких-либо скобок в результате вычисления выражения мы получим 14.

Листинг 2.22. Порядок вычисления операндов

```
class Program
{
    static int X(string label, int i)
    {
        Console.Write(label);
        return i;
    }

    static void Main(string[] args)
    {
        Console.WriteLine(X("a", 1) + X("b", 1) + X("c", 1) + X("d", 1));
    }
}
```

В примере определен метод `X`, который принимает два аргумента. Он отображает первый и просто возвращает второй. Затем я несколько раз использовал это в выражении, чтобы мы могли точно увидеть, когда вычисляются операнды, вызывающие `X`. Некоторые языки предпочитают не определять этот порядок, что делает поведение подобной программы непредсказуемым, но C# этот порядок определяет. Правило состоит в том, что в любом выражении операнды оцениваются в порядке их появления в исходном коде. Итак, когда выполняется вызов `Console.WriteLine` из листинга 2.22, он, в свою очередь, делает несколько вызовов `X`, которые каждый раз вызывают `Console.Write`, поэтому в качестве вывода мы видим: `abcd4`.

Однако здесь кроется важный нюанс: что именно мы подразумеваем под порядком выражений, когда происходит вложение? Весь аргумент этого `Console.WriteLine` представляет собой одно большое выражение сложения, где первым операндом является `X ("a", 1)`, а вторым — другое выражение сложения, которое, в свою очередь, имеет первым операндом `X ("b", 1)` и вторым — еще одно выражение сложения, операндами которого служат `X ("c", 1)` и `X ("d", 1)`. Если взять первое из этих выражений сложения, составляющее весь аргумент `Console.WriteLine`, имеет ли смысл спрашивать, следует оно до или после первого операнда? Лексически самое внешнее выражение сложения начинается точно в той же точке, где начинается его первый операнд, и заканчивается в точке, где заканчивается его второй операнд (что также происходит в той же самой точке, где заканчивается последний `X ("d", 1)`). В данном случае это не имеет значения, потому что единственным наблюдаемым эффектом порядка вычисления является результат, который выводит вызываемый метод `X`. Ни одно из выражений,

которые вызывают `X`, не вложено друг в друга, поэтому мы можем достоверно сказать, в каком порядке следуют эти выражения, и вывод, который мы видим, соответствует этому порядку. Однако в некоторых случаях (как в листинге 2.23) пересечение вложенных выражений может оказаться на результат видимое влияние.

Листинг 2.23. Порядок вычисления операнда с вложенными выражениями

```
Console.WriteLine(  
    X("a", 1) +  
    X("b", (X("c", 1) + X("d", 1) + X("e", 1))) +  
    X("f", 1));
```

Здесь аргумент `Console.WriteLine` складывает результаты трех вызовов `X`; однако второй из этих вызовов `X` (первый аргумент "`b`") принимает в качестве второго аргумента выражение, которое складывает результаты еще трех вызовов `X` (с аргументами "`c`", "`d`" и "`e`"). С последним вызовом `X` (передающим "`f`") в этом операторе у нас есть шесть выражений, вызывающих `X`. Правило C# для вычисления выражений в том порядке, в котором они появляются, применяется как обычно, но из-за частичного пересечения результаты способны поначалу удивить. Хотя в исходном коде буквы появляются в алфавитном порядке, вывод будет выглядеть как "`acdefbf5`". Если вам интересно, как вообще это может быть совместимо с вычислением выражений в порядке появления, учтите, что код начинает вычисление каждого выражения в порядке, в котором начинаются выражения, и заканчивает вычисление в том порядке, в котором выражения заканчиваются, но это два разных порядка. В частности, выражение, которое вызывает `X` с "`b`", начинает вычисляться перед теми, которые вызывают его с "`c`", "`d`" и "`e`", но заканчивается его вычисление уже после них. И это дает тот сбой в порядке следования, который мы видим в результате. Если вы найдете каждую закрывающую скобку, которая соответствует вызову `X` в этом примере, вы обнаружите, что порядок вызовов точно соответствует отображаемому.

Комментарии и пробелы

В большинстве языков программирования в исходных файлах может содержаться текст, который игнорируется компилятором, и C# не является исключением. Как и в большинстве языков семейства C, для этой цели поддерживаются два стиля комментариев. Есть односторонние комментарии,

как в листинге 2.24, в которых вы пишете подряд два символа `/`, и все, что идет дальше, игнорируется компилятором.

Листинг 2.24. Однострочные комментарии

```
Console.WriteLine("Say"); // Этот текст будет проигнорирован, но код слева  
Console.WriteLine("Anything"); // по-прежнему компилируется как обычно.
```

C# также поддерживает комментарии с разделителями. Вы начинаете комментарий с `/*`, и компилятор игнорирует последующее до тех пор, пока не встретит первую последовательность символов `*/`. Это может быть полезно, если вы не хотите, чтобы комментарий продолжался до конца строки, как это показано в первой строке листинга 2.25. Такой пример также показывает, что комментарии с разделителями могут занимать несколько строк.

Листинг 2.25. Комментарии с разделителями

```
Console.WriteLine(/* Has side effects */ GetLog());  
  
/* Некоторые разработчики любят использовать комментарии  
* с разделителями для больших блоков текста, где им нужно объяснить  
* что-то особенно сложное или странное в коде. Колонна звездочек  
* слева служит лишь для украшения – звездочки нужны только в начале  
* и в конце комментария.  
*/
```

Есть небольшая загвоздка, с которой вы можете столкнуться при работе с комментариями с разделителями; это может произойти, даже если комментарий находится в одной строке, но чаще происходит с многострочными комментариями. Листинг 2.26 показывает проблему с комментарием, который начинается в середине первой строки и заканчивается в конце четвертой.

Листинг 2.26. Многострочные комментарии

```
Console.WriteLine("This will run"); /* Этот комментарий включает  
Console.WriteLine("This won't");      * в себя не только текст в правой  
Console.WriteLine("Nor will this");   * части, но и текст в левой,  
Console.WriteLine("Nor this");        * за исключением первой  
Console.WriteLine("This will also run"); * и последней строк. */
```

Обратите внимание, что последовательность символов `/*` появляется в этом примере дважды. Когда эта последовательность появляется в середине комментария, она не делает ничего особенного — комментарии не вкладываются.

Несмотря на то что мы встретили две последовательности `/*`, первой же `*/` достаточно для завершения комментария. Такое может расстраивать, но для языков семейства С это норма.

Иногда полезно временно вывести часть кода из строя, чтобы потом его было легко вернуть. Превращение кода в комментарий является обычным способом добиться этого, и хотя использование комментариев с разделителями может показаться очевидным решением, возникают проблемы, если в область, которую вы закомментировали, вкрапляется еще один комментарий с разделителями. Поскольку вложения не поддерживаются, нужно добавить `/*` после закрытия внутреннего комментария `*/`, чтобы закомментировать весь диапазон. Поэтому для этой цели принято использовать однострочные комментарии. (Вы также можете использовать директиву `#if`, описанную в следующем разделе.)



Visual Studio может комментировать области кода за вас. Если вы выделите несколько строк текста и нажмете `Ctrl-K`, а затем сразу `Ctrl-C`, то это добавит `//` в начало каждой строки в выделении. Раскомментировать область можно с помощью `Ctrl-K`, `Ctrl-U`. Если при первом запуске Visual Studio в качестве предпочтительного языка вы выбрали что-то отличное от C#, описанные действия могут быть привязаны к другим последовательностям клавиш, но они также доступны в меню `Edit→Advanced`, а также на панели инструментов `Text Editor`, одной из стандартных панелей инструментов, которые Visual Studio показывает по умолчанию.

Говоря об игнорируемом тексте, нельзя не упомянуть, что C# по большей части игнорирует лишние пробелы. Не все пробелы незначительны, поэтому нужно хотя бы некоторое пространство для разделения синтаксических единиц, которые полностью состоят из буквенно-цифровых символов. Например, вы не можете написать `staticvoid` в качестве начала объявления метода — понадобится хотя бы один пробел (или символ табуляции, новая строка или другой похожий на пробел символ) между `static` и `void`. Но если речь не идет об алфавитно-цифровых синтаксических единицах, пробелы являются необязательными и в большинстве случаев один пробел эквивалентен любому количеству пробелов и новых строк. Это означает, что все три оператора в листинге 2.27 эквивалентны.

Листинг 2.27. Незначащие пробелы

```
Console.WriteLine("Testing");
Console . WriteLine("Testing");
Console.
    WriteLine ("Testing" )
;
```

Есть пара случаев, когда C# становится чувствителен к пробелам. Внутри строкового литерала пробел имеет значение, потому что все пробелы, которые вы пишете, будут присутствовать в строковом значении. Кроме того, C# обычно все равно, помещаете ли вы каждый элемент в свою собственную строку, или помещаете весь свой код в одну огромную строку, или (что наиболее вероятно) делаете что-то промежуточное, но есть и исключение: директивам предварительной обработки требуются собственные строки.

Директивы препроцессора

Если вы знакомы с языком С или его прямыми потомками, возможно, вам интересно, есть ли в C# препроцессор. У него нет отдельной стадии предварительной обработки и нет макросов. Тем не менее в нем имеется несколько директив, похожих на те, которые предлагает препроцессор С, хотя выбор их очень ограничен. Несмотря на то что в C# нет полноценной стадии предварительной обработки, как в С, они тем не менее называются директивами препроцессора.

Символы компиляции

В C# имеется директива `#define`, которая позволяет определить символ компиляции. Как правило, эти символы используются совместно с директивой `#if` для компиляции кода различными способами в различных ситуациях. Например, вам может понадобиться, чтобы какой-то код присутствовал только в сборках отладки, или для достижения определенного эффекта нужно использовать разный код на разных платформах. Но директиву `#define` не придется использовать слишком часто, так как более распространенный подход — это определение символов компиляции через настройки сборки компилятора. Visual Studio позволяет настраивать различные значения символов для каждой конфигурации сборки. Для этого щелкните правой кнопкой мыши на узле проекта в **Solution Explorer**, выберите **Properties** и на открывшейся странице свойств перейдите на вкладку **Build**. Также можно

просто открыть файл .csproj и определить желаемые значения в элементе `<DefineConstants>` любой `<PropertyGroup>`.



Определенные символы .NET SDK устанавливает по умолчанию. Он поддерживает две конфигурации, `Debug` и `Release`. Он определяет символ компиляции `DEBUG` в конфигурации `Debug`, тогда как в `Release` вместо него будет определен `RELEASE`. В обеих конфигурациях будет определен символ `TRACE`. Некоторые типы проектов получают дополнительные символы. Например, для библиотеки, ориентирующейся на .NET Standard 2.0, будут определены как `NETSTANDARD`, так и `NETSTANDARD2_0`.

Символы компиляции обычно используются вместе с директивами `#if`, `#else`, `#elif` и `#endif`. (`#elif` – это сокращение от `else if`.) В листинге 2.28 используются некоторые из этих директив, чтобы гарантировать, что определенные строки кода компилируются только в сборках `Debug`. (Вы также можете написать `#if false`, чтобы полностью предотвратить компиляцию фрагментов кода. Обычно это служит временной мерой и является альтернативой комментированию, которое обходит некоторые из лексическихловушек, связанных с вложенными комментариями.)

Листинг 2.28. Условная компиляция

```
#if DEBUG
    Console.WriteLine("Starting work");
#endif
    DoWork();
#endif DEBUG
    Console.WriteLine("Finished work");
#endif
```

C# предоставляет более тонкий механизм для такого рода вещей, называемый условным методом. Компилятор распознает атрибут, определенный библиотеками классов .NET, под названием `ConditionalAttribute`, для которого он обеспечивает специальное поведение во время компиляции. Вы можете аннотировать этим атрибутом любой метод. В листинге 2.29 он используется для указания того, что аннотированный метод должен применяться только тогда, когда определен символ компиляции `DEBUG`.

Если вы напишете код, который вызывает аннотированный таким образом метод, компилятор C# пропустит этот вызов в сборках, которые не

определяют соответствующий символ. Поэтому, если вы напишете код, который вызывает метод `ShowDebugInfo`, компилятор удалит эти вызовы в сборках, отличных от `Debug`. Это означает, что можно добиться такого же эффекта, что и в листинге 2.28, и при этом не загромождать свой код директивами.

Листинг 2.29. Условный метод

```
[System.Diagnostics.Conditional("DEBUG")]
static void ShowDebugInfo(object o)
{
    Console.WriteLine(o);
}
```

Классы `Debug` и `Trace` библиотеки классов .NET в пространстве имен `System.Diagnostics` используют эту функцию. Класс `Debug` предлагает различные методы, которые являются условными для символа компиляции `DEBUG`, в то время как класс `Trace` содержит методы, условные для `TRACE`. Если оставить для нового проекта C# настройки по умолчанию, любой диагностический вывод, произведенный с помощью класса `Trace`, будет доступен в сборках `Debug` и `Release`, но любой код, вызывающий метод класса `Debug`, не будет скомпилирован в сборке `Release`.



Метод `Assert` класса `Debug` зависит от `DEBUG`, что иногда застает разработчиков врасплох. `Assert` позволяет вам задать условие, которое должно быть истинным во время выполнения, и выдает исключение, если условие ложно. Разработчики, плохо знакомые с C#, часто ошибочно помещают в `Debug.Assert` проверки, которые должны выполняться во всех сборках, а также выражения с побочными эффектами, от которых зависит остальная часть кода. Это приводит к ошибкам, потому что компилятор удалит этот код в сборках, отличных от `Debug`.

#error и #warning

C# позволяет генерировать ошибки или предупреждения компилятора с помощью директив `#error` и `#warning`. Как правило, они используются внутри условных областей, как это показано в листинге 2.30. Хотя безусловный `#warning` может быть полезен для напоминания о том, что вы еще не написали какой-то особо важный фрагмент кода.

Листинг 2.30. Генерация ошибки компилятора

```
#if NETSTANDARD
    #error .NET Standard is not a supported target for this source file
#endif
```

#line

Директива `#line` полезна в сгенерированном коде. Когда компилятор выдает ошибку или предупреждение, он обычно указывает, где возникла проблема, указывая имя файла, номер строки и смещение в этой строке. Но если рассматриваемый код был создан автоматически с использованием какого-либо другого файла в качестве входных данных и если этот другой файл содержит основную причину проблемы, может быть более полезно сообщить об ошибке во входном, а не в сгенерированном файле. Директива `#line` может дать указание компилятору C# действовать так, как если бы ошибка произошла в строке с указанным номером и, что необязательно, как если бы ошибка была в совершенно другом файле. Листинг 2.31 показывает, как ее использовать. При использовании директивы об ошибке будет сообщаться так, как будто она произошла в строке 123 файла с именем `Foo.cs`.

Листинг 2.31. Директива `#line` и преднамеренная ошибка

```
#line 123 "Foo.cs"
intt x;
```

Часть с именем файла является необязательной, что позволяет подделывать только номера строк. Вы можете дать команду компилятору вернуться к сообщениям о предупреждениях и ошибках без подделки, написав `#line default`.

Эта директива также влияет и на отладку. Когда компилятор выдает отладочную информацию, он учитывает директивы `#line`. Это означает, что при просмотре кода в отладчике вы увидите место, на которое ссылается `#line`.

У этой директивы есть и другое использование. Вместо номера строки (и необязательного имени файла) вы можете написать просто `#line hidden`. Это влияет только на поведение отладчика: при пошаговом выполнении Visual Studio будет выполнять весь код после такой директивы без остановки, пока не встретит нескрытую директиву `#line` (обычно это `#line default`).

#pragma

Директива `#pragma` выполняет две функции: ее можно использовать для отключения выбранных предупреждений компилятора, а также для переопределения значений контрольной суммы, которые компилятор помещает в генерируемый им файл `.pdb`, содержащий отладочную информацию. Обе они предназначены в основном для случаев генерации кода, хотя иногда могут оказаться полезны и для отключения предупреждений в обычном коде. В листинге 2.32 показано, как использовать `#pragma` для предотвращения выдачи компилятором предупреждения, которое обычно возникает, если вы объявляете переменную, которую затем не используете.

Листинг 2.32. Отключение предупреждения компилятора

```
#pragma warning disable CS0168  
    int a;
```

Рекомендуется избегать отключения предупреждений. Эта функция полезна в сгенерированном коде, потому что генерация кода может в конечном итоге создавать элементы, которые не всегда используются, и директива `#pragma` — единственный способ получить чистую компиляцию. Но, когда вы пишете код вручную, как правило, от предупреждений можно избавиться обычным путем.

Некоторые компоненты NuGet предоставляют анализаторы кода, компоненты, которые подключаются к API компилятора C# и которым предоставляется возможность проверять код и генерировать собственные диагностические сообщения. (Это происходит во время сборки, а в Visual Studio это происходит еще и во время редактирования, что обеспечивает диагностику непосредственно при наборе кода. Они также работают в реальном времени в коде Visual Studio, если включить расширение `OmniSharp C#`.) Например, пакет NuGet `StyleCop.Analyzers` предоставляет анализатор, который предупредит вас, если какие-либо публичные члены вашего типа не соответствуют руководству по проектированию библиотеки классов Microsoft. Используйте директивы `#pragma warning` для управления предупреждениями от анализаторов кода, а не только от компилятора C#. Обычно анализаторы ставят перед номерами своих предупреждений несколько букв, чтобы вы могли их отличать, — например, предупреждения компилятора начинаются с `CS`, а `StyleCop` — с `SA`.

Прагмы делают возможной обработку предупреждений, которые вы получаете при использовании ссылок с возможным нулевым значением,

добавленных в C# 8.0. Вместо указания номера предупреждения, которое генерирует компилятор или анализатор кода, вы можете написать `nullable` (например, `#pragma warning disable nullable`). Подробности см. в главе 3.

Вполне возможно, что в будущих версиях C# могут добавиться другие функции, основанные на `#pragma`. Когда компилятор встречает прагму, которую не понимает, то генерирует предупреждение, а не ошибку на том основании, что нераспознанная прагма может быть допустимой для какой-либо будущей версии компилятора или компилятора другого производителя.

#nullable

В C# 8.0 добавилась новая директива `#nullable`, которая позволяет тонко контролировать контекст аннотаций, допускающих значение `null`. Это часть функционала, касающегося ссылочных типов, допускающих значение `null`, о котором пойдет речь в главе 3. (Она не пересекается с управлением предупреждениями о возможном значении `null`, описанном в предыдущем разделе, потому что мы имеем возможность контролировать как независимое включение аннотаций о допущении значения `null`, так и включение предупреждений, связанных с этими аннотациями.)

#region и #endregion

Наконец, у нас есть две директивы предварительной обработки, которые ничего не делают. Если вы напишете директиву `#region`, то единственное, что сделает компилятор, — убедится, что она имеет соответствующую директиву `#endregion`. Несоответствие вызовет ошибку компилятора, но правильные пары директив `#region` и `#endregion` компилятор игнорирует. Области, задаваемые этими директивами, могут быть вложенными.

Эти директивы существуют исключительно ради текстовых редакторов, которые умеют их распознавать. Visual Studio использует их для предоставления возможности сворачивания областей кода в одну строку. Редактор C# автоматически позволяет разворачивать и сворачивать определенные функции, такие как определения классов, методы и блоки кода (функция называется структуризацией). Если вы определите области с помощью этих двух директив, это также позволит их разворачивать и сворачивать. Область можно выделить как в более мелком отрывке кода (например, в пределах одного блока), так и в более крупном (например, в нескольких связанных методах), чем те, что редактор предлагает автоматически.

Если навести курсор мыши на свернутую область, Visual Studio отобразит всплывающую подсказку, отображающую содержимое области. После маркера `#region` можно поместить текст. Когда среда Visual Studio отображает свернутую область, она показывает этот текст в той единственной строке, которая остается. Несмотря на то что вы можете опустить эту часть, как правило, включение какого-то описательного текста — это хорошая идея, так как люди смогут иметь лишь приблизительное представление о том, что увидят, если расширят такую область.

Некоторым нравится помещать все содержимое класса в разные области, потому что, свернув их все, вы можете мгновенно увидеть структуру файла. Она может даже уместиться на экране, благодаря тому что области сводятся к единственной строке. С другой стороны, некоторые люди терпеть не могут свернутые области, потому что те препятствуют изучению кода, а также могут породить привычку помещать слишком много исходного кода в один файл.

Основные типы данных

.NET определяет тысячи типов в своей библиотеке классов, а вы можете написать свой собственный, поэтому C# способен работать с неограниченным количеством типов данных. Тем не менее несколько типов обрабатываются компилятором особым образом. Ранее, в листинге 2.9, вы видели, что если у вас имеется строка и вы пытаетесь добавить к ней число, результирующий код преобразует число в строку и добавляет ее к исходной строке. На самом деле такое поведение имеет более общий характер, не ограничиваясь лишь числами. Скомпилированный код вызовет метод `String.Concat`, и если вы передадите ему любые нестроковые аргументы, перед конкатенацией он вызовет их методы `ToString`. Метод `ToString` есть у всех типов, а это означает, что вы можете добавлять к строке значения любого типа.

Это удобно, но работает лишь потому, что компилятор C# знает о строках и обрабатывает их особым образом. (Часть спецификации C# описывает эту специфичную обработку строк в случае оператора `+`.) C# предоставляет различные специальные сервисные функции не только для строк, но и для определенных числовых типов данных, логических значений, семейства типов, называемых кортежами, а также двух конкретных типов, `dynamic` и `object`. Большинство из них являются особенными не только для C#, но и для среды выполнения — почти все числовые типы (кроме `BigInteger`) на-

прямую поддерживаются промежуточным языком (IL), а типы `bool`, `string` и `object` имеют еще и внутреннюю реализацию в среде выполнения.

Числовые типы

C# поддерживает целочисленную и арифметику с плавающей точкой. Существуют целые типы со знаком и без знака, и они бывают разных размеров, как показано в табл. 2.1. Наиболее часто используемый целочисленный тип — это `int`, не в последнюю очередь из-за того, что он достаточно большой, чтобы представлять широкий диапазон значений, но не слишком большой, что позволяет эффективно работать на всех процессорах, поддерживающих .NET. (Большие типы данных могут не обрабатываться непосредственно ЦП и иметь нежелательные особенности при работе в многопоточном коде: чтение и запись являются атомарными для 32-битных типов, но могут не быть таковыми для более крупных.)⁵

Таблица 2.1. Целочисленные типы

Тип в C#	Имя в CLR	Со знаком	Размер, битов	Диапазон
байт	System.Byte	Нет	8	от 0 до 255
sbyte	System.SByte	Да	8	от -128 до 127
ushort	System.UInt16	Нет	16	от 0 до 65,535
short	System.Int16	Да	16	от -32,768 до 32,767
uint	System.UInt32	Нет	32	от 0 до 4,294,967,295
int	System.Int32	Да	32	от -2,147,483,648 до 2,147,483,647
ulong	System.UInt64	Нет	64	от 0 до 18,446,744,073,709,551,615
long	System.Int64	Да	64	от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807

Второй столбец в табл. 2.1 показывает имя типа в CLR. Различные языки имеют разные соглашения об именах, и для числовых типов C# использует имена, обусловленные его происхождением из семейства языков C. Но они не

⁵ Строго говоря, это гарантировано только для 32-битных типов с корректным выравниванием. Однако C# корректно выравнивает их по умолчанию, так что вы, как правило, имеете дело с невыровненными данными, только если ваш код вынужден обращаться к неуправляемому коду.

соответствуют соглашениям об именах типов данных, принятым в .NET. Что касается среды выполнения, имена во втором столбце являются реальными именами — существуют различные API, которые способны сообщать информацию о типах во время выполнения, и они сообщают именно эти имена CLR, а не имена C#. Эти имена синонимичны в исходном коде C#, поэтому, если есть желание, можно свободно использовать имена среды выполнения, но имена C# лучше соответствуют стилю, так как все ключевые слова в языках семейства С — строчные. Поскольку компилятор обрабатывает эти типы не так, как остальные, возможно, и хорошо, что они выделяются.



Не все языки .NET поддерживают числа без знака, поэтому библиотека классов .NET стремится избегать их использования. Среда выполнения, поддерживающая несколько языков (например, CLR), сталкивается с необходимостью компромисса в том, чтобы предложить систему типов, достаточно богатую для удовлетворения потребностей большинства языков, и при этом не принуждать слишком простые языки использовать чрезмерно сложную систему типов. Чтобы решить эту проблему, система типов .NET, CTS, является всеобъемлющей, но языки не обязаны поддерживать ее целиком. CLS определяет относительно небольшое подмножество CTS, которое должно поддерживаться всеми языками. Целые числа со знаком входят в CLS, а целые числа без знака — нет. Вот почему вы иногда будете встречать неожиданные варианты типов, такие как свойство `Length` массива, являющееся `int` (а не `uint`), несмотря на тот факт, что оно никогда не вернет отрицательное значение.

C# также поддерживает числа с плавающей точкой. Существует два типа: `float` и `double`, которые представляют собой 32-битные и 64-битные числа в стандартных форматах IEEE 754, и, как показывают названия CLR в табл. 2.2, они соответствуют тому, что обычно называют числами с одинарной и двойной точностью. Значения с плавающей точкой работают не так, как целые числа, поэтому эта таблица немного отличается от таблицы целочисленных типов. Числа с плавающей запятой хранят значение и показатель степени (по концепции похожи на научные обозначения, но работают в двоичном исчислении, а не в десятичном). Столбец точности показывает, сколько битов доступно для значения, а затем показан диапазон в виде наименьшего ненулевого значения и наибольшего значения, которое может быть им представлено. (Они могут быть как положительными, так и отрицательными.)

Таблица 2.2. Типы с плавающей точкой

Тип в C#	Имя в CLR	Размер, битов	Точность	Диапазон (величина)
<code>float</code>	<code>System.Single</code>	32	23 бита (~7 десятичных цифр)	от 1.5×10^{-45} до 3.4×10^{38}
<code>double</code>	<code>System.Double</code>	64	52 бита (~15 десятичных цифр)	от 5.0×10^{-324} до 1.7×10^{308}

C# распознает и третье числовое представление, называемое `decimal` (или `System.Decimal` в CLR). Это 128-битное значение, поэтому оно способно дать большую точность, чем другие форматы, но это не просто увеличенная версия `double`. Формат предназначен для вычислений, которые требуют предсказуемой обработки десятичных дробей, чего не могут предложить ни `float`, ни `double`. Если вы пишете код, который инициализирует переменную типа `float` в `0`, а затем девять раз подряд добавляет к ней `0.1`, то ожидаете получить значение `0.9`, но на самом деле получаете приблизительно `0.9000001`. Это связано с тем, что IEEE 754 хранит числа в двоичном формате, который не способен представлять все десятичные дроби. С некоторыми он справляется. Например, десятичное `0.5`, если записать его с основанием `2`, превращается в `0.1`. Но десятичное `0.1` в двоичном формате представляет собой периодическую дробь. (В частности, за `0.0` идет повторяющаяся последовательность `0011`.) Это означает, что `float` и `double` могут дать лишь приближение десятичной дроби `0.1`. А в более общем случае только небольшое подмножество десятичных чисел может быть представлено точно. Это не всегда очевидно, потому что когда числа с плавающей точкой преобразуются в текст, они округляются до десятичного приближения, которое способно замаскировать расхождение. Но в процессе многочисленных вычислений неточности, как правило, суммируются, что в конечном итоге приводит к неожиданным результатам.

Для некоторых видов расчетов это не имеет значения; например, при моделировании или обработке сигнала вполне ожидаемы некоторый шум и ошибки. Но счетоводы и финансовые регуляторы, как правило, менее снисходительны — небольшие несоответствия могут создать впечатление, что деньги волшебным образом исчезли или появились. При подсчете денег нужна абсолютная точность, что делает плавающую точку ужасным выбором для такой задачи. Вот почему C# предлагает тип `decimal`, который обеспечивает четко определенный уровень десятичной точности.

`Decimal` хранит числа в виде знакового бита (положительного или отрицательного) и пары целых чисел. Значением десятичного числа является

первое целое 96-битное число (отрицательное, если об этом говорит знаковый бит), умноженное на 10 в степени второго целого числа, лежащего в диапазоне от 0 до 28. 96 бит достаточно для представления любого 28-значного десятичного целого числа (и некоторых, но не всех 29-значных), поэтому второе целое число — то, которое представляет степень 10, — по сути указывает, где расположена десятичная точка⁶. Этот формат позволяет с точностью представить любое десятичное число с 28 или менее знаками.



Большинство целочисленных типов могут обрабатываться непосредственно процессором. (А в 64-битном процессе вообще все.) Кроме того, многие процессоры могут работать напрямую с представлениями `float` и `double`. Однако ни один из них не имеет встроенной поддержки `decimal`, а это означает, что даже простые операции, такие как сложение, требуют нескольких инструкций процессора. Это, в свою очередь, означает, что операции с `decimal` будут проходить гораздо медленнее, чем с другими числовыми типами, о которых мы говорили до сих пор.

Когда вы пишете числовое значение, то можете выбрать тип самостоятельно или позволить компилятору выбрать наиболее подходящий. Если вы напишите простое целое число, такое как 123, его тип будет `int`, `uint`, `long` или `ulong` — компилятор выбирает первый тип из этого списка с диапазоном, который содержит нужное значение. (Таким образом, 123 будет представлено `int`, 3000000000 — `uint`, 5000000000 — `long` и т. д.) Если вы напишите число с десятичной точкой, например 1.23, его тип будет `double`.

Если вы имеете дело с большими числами, очень легко ошибиться в подсчете нулей. В этом нет ничего хорошего, а в зависимости от области применения такая ошибка может стать дорогой или опасной. C# дает некоторое послабление, позволяя добавлять подчеркивания в любом месте числовых литералов, чтобы разбить числа на фрагменты по вашему усмотрению. Это аналогично общепринятой в большинстве англоязычных стран практике использования запятой для разделения нулей на группы по 3. Например, вместо того чтобы писать 5000000000, большинство носителей английского

⁶ Следовательно, десятичная дробь не использует все свои 128 бит. Уменьшение его размера вызовет трудности с выравниванием, а использование дополнительных битов для увеличения точности окажет значительное влияние на производительность, поскольку целые числа, длина которых кратна 32 битам, процессоры обрабатывают быстрее, чем альтернативные варианты.

языка написали бы 5,000,000,000, что позволяет гораздо легче определить, что это 5 миллиардов, а не, скажем, 50 миллиардов или 500 миллионов. (Многие носители английского языка не знают, что в ряде стран мира для этого используется точка, и они пишут 5.000.000.000, а запятую используют там, где большинство носителей английского языка поставят десятичную точку. Интерпретация стоимости, такой как € 100.000, требует, чтобы вы знали, соглашения какой страны используются, если, конечно, не хотите совершить катастрофический финансовый просчет. Но я отвлекся.⁷⁾) В C# мы можем делать нечто подобное, записывая числовой литерал в виде 5_000_000_000.

Вы можете с помощью суффикса указать компилятору, что вам нужен определенный тип. Таким образом, 123U — это `uint`, 123L — это `long`, а 123UL — это `ulong`. Буквы суффикса не зависят от регистра и порядка, поэтому вместо 123UL можно написать 123Lu, 123uL или любой другой вариант перестановки. Для `double`, `float` и `decimal` используйте суффиксы D, F и M соответственно.

Последние три типа поддерживают десятичный экспоненциальный литеральный формат для больших чисел, в котором буква E помещается в константу, за которой следует степень. Например, значение числового литерала `1.5E-20` — это 1.5, умноженные на 10 в степени -20. (Это тип `double`, потому что это тип по умолчанию для числа с десятичной точкой независимо от того, находится ли оно в экспоненциальном формате. Вы могли бы записать константы с эквивалентными значениями типа `float` и `decimal` как `1.5E-20F` и `1.5E-20M`.)

Часто полезно иметь возможность записывать целочисленные литералы в шестнадцатеричном формате, потому что эти знаки лучше соответствуют двоичному представлению, которое используется во время выполнения. Это особенно важно, когда разные битовые диапазоны числа представляют разные вещи. Например, вам, возможно, придется иметь дело с числовым кодом ошибки, источником которого служит системный вызов Windows, — они иногда встречаются в исключениях. В некоторых случаях эти коды используют самый верхний бит, чтобы указать на успех или сбой, следующие несколько битов — для указания на источник ошибки, а оставшиеся биты — для указания на конкретную ошибку. Например, код ошибки `COM_E_ACCESSDENIED -2,147,024,891`. Трудно увидеть структуру в десятичном формате, но в шестнадцатеричном все проще: `80070005`. 8 указывает на то, что это ошибка, а `007`, которая следует после восьмерки, указывает, что

⁷⁾ В российской традиции принято разделять разряды пробелами: 5 000 000 000. — Примеч. ред.

изначально это была простая ошибка Win32, которая была преобразована в ошибку COM. Остальные биты говорят о том, что код ошибки в Win32 был 5 (ERROR_ACCESS_DENIED). C# позволяет записывать целочисленные литералы в шестнадцатеричном формате для сценариев, где шестнадцатеричное представление более наглядно. Вы просто помещаете перед числом 0x, соответственно, в нашем случае вы должны написать 0x80070005.

Вы также можете записывать двоичные литералы, используя префикс 0b. Разделители цифр могут использоваться в шестнадцатеричном и двоичном виде точно так же, как и в десятичных числах, хотя в этих случаях цифры чаще группируются по четыре, например 0b_0010_1010. Очевидно, что это делает любую двоичную структуру в числе еще более наглядной, чем в шестнадцатеричном формате. Однако 32-битные двоичные литералы имеют не самую удобную длину, поэтому мы часто используем вместо них шестнадцатеричные.

Числовые преобразования

Каждый из встроенных числовых типов для хранения чисел в памяти использует собственное представление. Преобразование из одной формы в другую требует некоторой работы — даже число 1 будет выглядеть совершенно по-разному, если вы изучите его двоичные представления в виде float, int и decimal. Тем не менее C# может генерировать код, который конвертирует форматы друг в друга, и чаще всего это происходит автоматически. Листинг 2.33 показывает несколько таких случаев.

Листинг 2.33. Неявные преобразования

```
int i = 42;
double di = i;
Console.WriteLine(i / 5);
Console.WriteLine(di / 5);
Console.WriteLine(i / 5.0);
```

Вторая строка присваивает значение переменной int переменной double. Компилятор C# генерирует необходимый код для преобразования целочисленного значения в эквивалентное значение с плавающей точкой. Уже не так очевидно, но последние две строки выполняют аналогичные преобразования, что видно из вывода:

```
8
8.4
8.4
```

Видно, что первое деление дало целочисленный результат — деление целочисленной переменной `i` на целочисленный литерал 5 заставило компилятор генерировать код, который выполняет целочисленное деление, поэтому результат равен 8. Но два других деления вернули результат с плавающей запятой. Во втором случае мы разделили переменную типа `double`, названную `d1`, на целочисленный литерал 5. Перед выполнением деления C# преобразовал его в число с плавающей запятой. И в заключительной строке мы делим целочисленную переменную на литерал с плавающей точкой. На этот раз в значение с плавающей точкой перед делением конвертируется значение переменной.

В общем, когда вы выполняете арифметические вычисления, которые включают в себя смесь числовых типов, перед вычислением C# выберет тип с наибольшим диапазоном и переведет значения типов с более узким диапазоном в тип с большим. (Арифметические операторы обычно требуют, чтобы все их операнды имели одинаковый тип, поэтому, если вы предоставляете операнды разных типов, один тип должен «победить» для любого конкретного оператора.) Например, `double` может представлять любое значение, которое может `int`, и многие другие, поэтому `double` — более многозначный тип⁸.

C# неявно выполнит числовые преобразования всякий раз, когда преобразование является повышением (т. е. целевой тип имеет более широкий диапазон, чем исходный), потому что нет никакой вероятности сбоя такого преобразования. Тем не менее он не станет неявно преобразовывать в обратном направлении. Вторую и третью строки листинга 2.34 не удастся скомпилировать, поскольку они пытаются присвоить выражения типа `double` переменной типа `int`, что является сужающим преобразованием, т. е. источник может содержать значения, выходящие за пределы целевого диапазона.

Листинг 2.34. Ошибки: неявные преобразования недоступны

```
int i = 42;
int willFail = 42.0;
int willAlsoFail = i / 1.0;
```

⁸ Продвижение типов на самом деле не является функцией C#. Для этого существует более общепринятый механизм: операторы преобразования. C# определяет собственные операторы неявного преобразования для встроенных типов данных. Обсуждаемое здесь продвижение типов происходит в результате того, что компилятор следует своим обычным правилам преобразования.

Конвертация в этом направлении возможна, просто она должна быть явной. Вы можете использовать приведение, где указывается имя типа, в который вы хотите преобразовать содержимое скобок. Листинг 2.35 показывает модифицированную версию листинга 2.34, где мы явно заявляем, что хотим преобразовать `int`. При этом мы либо не возражаем, что данное преобразование может сработать неправильно, либо у нас есть основания полагать, что в этом конкретном случае значение останется в диапазоне. Обратите внимание, что в последней строке я поместил выражения после приведения в скобки. Таким образом приведение применяется ко всему выражению; в противном случае правила приоритета C# диктуют, что оно будет применяться только к переменной `i`, и поскольку она уже и так типа `int`, это не будет иметь никакого эффекта.

Листинг 2.35. Явные преобразования с приведениями

```
int i = 42;
int i2 = (int) 42.0;
int i3 = (int) (i / 1.0);
```

Таким образом, сужающие преобразования требуют явного приведения, а преобразования, которые не способны привести к потере информации, происходят неявно. Однако с некоторыми комбинациями типов ни один не является явно более предпочтительным. Что произойдет, если вы попытаетесь прибавить `int` к `uint`? Или `int` к `float`? Все эти типы имеют размер 32 бита, поэтому ни один из них не может предложить больше 2^{32} различных значений. Вместе с тем они имеют разные диапазоны, что говорит, что у каждого есть значения, которые он представлять может, а другие типы — нет. Например, вы можете представить значение 3,000,000,001 в формате `uint`, но оно слишком велико для типа `int` и может быть лишь приближенным в случае `float`. По мере того как числа с плавающей точкой становятся больше, значения, которые могут быть последовательно представлены, становятся все дальше друг от друга. Число с плавающей точкой может представлять 3,000,000,000, а также 3,000,001,024, но ничего между ними. Таким образом, для значения 3 000 000 001 `uint` кажется лучшим выбором, чем `float`. Но как насчет -1? Это отрицательное число, поэтому типу `uint` оно не по зубам. Кроме того, есть очень большие числа, которые могут быть представлены в виде `float`, но находятся вне диапазона как `int`, так и `uint`. Каждый из перечисленных типов имеет свои сильные и слабые стороны, и нет смысла говорить, что один из них в целом лучше, чем остальные.

Удивительно, но C# допускает некоторые неявные преобразования даже в сценариях с возможными потерями. Правила учитывают только диапазон, а не точность: неявные преобразования разрешены, если диапазон целевого типа полностью содержит диапазон исходного типа. Таким образом, вы можете конвертировать из `int` или `uint` в `float`, потому что, хотя `float` не может точно представить некоторые значения, нет значений `int` или `uint`, которые он не может представить хотя бы приближенно. Но неявные преобразования не допускаются в другом направлении, потому что есть некоторые значения `float`, которые попросту слишком велики, — в отличие от `float`, целочисленные типы не могут представлять приближения для больших чисел.

Вам может быть интересно, что произойдет, если произвести сужающее преобразование `int` с помощью приведения, как в листинге 2.35, в ситуации, когда число выходит за пределы допустимого диапазона. Ответ зависит от типа, из которого вы преобразовываете. Преобразование из одного целочисленного типа в другой работает иначе, чем преобразование из числа с плавающей запятой в целое. На самом деле спецификация C# не определяет, каким образом слишком большие числа с плавающей точкой должны быть преобразованы в целочисленный тип, т. е. результатом может оказаться что угодно. Но при приведении между целочисленными типами результат определен четко. Если два типа имеют разные размеры, двоичное представление будет либо обрезано, либо дополнено нулями (или единицами, если тип со знаком, а значение отрицательно), чтобы сделать его подходящего размера для целевого типа. После этого биты обрабатываются так, как если бы они представляли целевой тип. Иногда это полезно, но чаще дает неожиданные результаты, поэтому вы можете выбрать альтернативное поведение для любого приведения вне диапазона, сделав такое преобразование проверенным.

Проверенные контексты

В C# определено ключевое слово `checked`, которое вы можете поместить перед оператором блока или выражением, сделав его проверенным контекстом. Это означает, что определенные арифметические операции, включая приведение, во время выполнения проверяются на арифметическое переполнение. Если вы приведете значение к целочисленному типу в проверенном контексте, а значение слишком большое или маленькое, произойдет ошибка — код вызовет исключение `System.OverflowException`.

Помимо проверки приведений проверенный контекст будет следить и за переполнением диапазона при обычных вычислениях. Сложение, вычитание

и другие операции могут принимать значения за пределами диапазона их типа данных. Для целых чисел это приводит к тому, что если значение не проверять, то оно «переворачивается», поэтому добавление 1 к максимальному значению дает минимальное значение и наоборот в случае вычитания. Так что иногда эта обертка может оказаться полезной. Например, если вы хотите определить, сколько времени прошло между двумя точками в коде, один из способов проверить это — свойство `Environment.TickCount`. (Это более надежно, чем использование текущей даты и времени, потому что они могут измениться в результате настройки часов или при перемещении между часовыми поясами⁹. Количество тиков возрастает с постоянной скоростью. Но все же в реальном коде вы, скорее всего, использовали бы класс `Stopwatch` библиотеки классов.) Листинг 2.36 показывает один из способов такого использования.

Листинг 2.36. Использование непроверенного целочисленного переполнения

```
int start = Environment.TickCount;  
DoSomeWork();  
int end = Environment.TickCount;  
  
int totalTicks = end - start;  
Console.WriteLine(totalTicks);
```

Особенность `Environment.TickCount` заключается в том, что он время от времени «обворачивается». Он подсчитывает количество миллисекунд с момента последней перезагрузки системы, и поскольку его тип `int`, в конечном итоге он выходит за пределы допустимого диапазона. Интервал в 25 дней — это 2,16 миллиарда миллисекунд, слишком большое число, чтобы поместиться в `int`. (.NET Core 3.0 решает эту проблему с помощью свойства `TickCount64`, что работает для почти 300 миллионов лет. Но оно недоступно в более старых версиях или в любой версии .NET Standard, актуальной на момент написания этой статьи.) Представьте себе, что число тиков составляет 2 147 483 637, что на 10 меньше максимального значения для `int`. Что, по-вашему, должно произойти через 100 мс? Число не может просто стать на 100 больше (2 147 483 727), потому что это будет слишком большим значением для `int`. Следует ожидать, что оно достигнет максимально возможного значения уже через 10 мс, поэтому через 11 мс

⁹ Свойство является членом типа, представляющего значение, которое может быть прочитано или изменено, или же и то и другое сразу; глава 3 описывает свойства более подробно.

оно будет возвращено к минимальному значению; таким образом, через 100 мс ожидаемое число тиков будет на 89 выше минимального значения (что будет равно $-2,147,483,559$).



На практике число тиков не обязательно имеет точность до миллисекунды. Счетчик часто замирает на несколько миллисекунд, а затем прыгает вперед с шагом 10, 15 мс или больше. Однако значение по-прежнему вернется к минимальному, просто вы при этом не сможете наблюдать все возможные значения счетчика.

Примечательно, что листинг 2.36 отлично справляется с такой ситуацией. Если значение счетчика в `start` было получено незадолго до оборота, а значение в `end` — сразу после, `end` будет содержать намного меньшее значение, чем `start`, и покажется как будто перевернутым, а разница между ними будет очень большой — больше, чем диапазон `int`. Однако когда мы вычитаем `start` из `end`, переполнение оборачивает все таким образом, который точно соответствует способу оборота счетчика тиков, что, в свою очередь, означает, что в конечном итоге мы получим верный результат. Например, если `start` содержит счетчик тиков за 10 мс до оборота, а `end` — через 90 мс после, вычитание соответствующих счетчиков (т. е. вычитание $-2,147,483,558$ из $2,147,483,627$) очевидно должно привести к результату $4,294,967,185$. Но из-за того, что при вычитании происходит переполнение, по факту мы получаем 100, что соответствует затраченным 100 мс.

Но в большинстве случаев такое целочисленное переполнение нежелательно. Все это означает, что при работе с большими числами вы можете столкнуться с совершенно неверными результатами. В большинстве случаев риск невелик, потому что вы будете иметь дело с довольно небольшими числами, но, если существует вероятность переполнения во время вычислений, возможно, стоит задуматься о том, чтобы использовать проверенный контекст. Любые арифметические операции, выполняемые в проверенном контексте, при переполнении будут вызывать исключение. Вы можете задать такое поведение выражения с помощью оператора `checked`, как показано в листинге 2.37. Все, что расположено в скобках, будет вычисляться в проверенном контексте, поэтому вы увидите `OverflowException`, если при сложении `a` и `b` произойдет переполнение. Ключевое слово `checked` здесь относится не ко всему выражению, поэтому если в результате прибавления с произойдет переполнение, это не вызовет исключения.

Листинг 2.37. Проверенное выражение

```
int result = checked(a + b) + c;
```

С помощью оператора `checked` вы также можете включить проверку всего блока кода, поместив ключевое слово `checked` перед самим блоком, как показано в листинге 2.38. Операторы `checked` всегда включают какой-то блок, так что вы не можете просто добавить ключевое слово `checked` перед ключевым словом `int` в листинге 2.37, чтобы превратить его в проверенный оператор. Кроме того, необходимо заключить код в фигурные скобки.

Листинг 2.38. Проверенный оператор

```
checked
{
    int r1 = a + b;
    int r2 = r1 - (int) c;
}
```



Оператор `checked` затрагивает только строки кода внутри блока. Если код вызывает какие-либо посторонние методы, то на них ключевое слово `checked` уже не повлияет, так как в CPU нет специального бита `checked`, который включается для текущего потока внутри проверенного блока. (Другими словами, область действия этого ключевого слова является лексической, а не динамической.)

В C# также имеется ключевое слово `unchecked`. Его можно использовать внутри проверенного блока, чтобы указать, что конкретное выражение или вложенный блок не должны быть проверенным контекстом. Это облегчает жизнь, если вы хотите, чтобы проверялось все, кроме единственного конкретного выражения. Вместо того чтобы помечать все, кроме выбранной части, как проверенное, можно поместить весь код в проверенный блок и затем исключить один фрагмент, в котором допустимо переполнение без того, чтобы вызывать ошибку.

Вы можете так настроить компилятор C#, чтобы по умолчанию все помещалось в проверенный контекст и только явно помеченные `unchecked` выражения и операторы могли молча переполняться. В Visual Studio это можно настроить, открыв свойства проекта, перейдя на вкладку `Build` и нажав кнопку `Advanced`. Еще можно отредактировать файл `.csproj`, добавив `<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>` внутрь `<PropertyGroup>`. Имейте в виду, что за это придется заплатить заметную

цену — проверка может в несколько раз замедлить отдельные целочисленные операции. Влияние на ваше приложение в целом будет меньше, потому что программы, как правило, не заняты лишь арифметическими операциями, но все же цена может оказаться ненулевой. Конечно, как и в любом другом вопросе, касающемся производительности, необходимо оценить реальное влияние. Вы можете обнаружить, что понижение производительности является приемлемой ценой гарантии того, что вы вовремя узнаете о непредвиденных переполнениях.

BigInteger

Есть еще один числовой тип, о котором стоит знать: **BigInteger**. Он является частью библиотеки классов .NET и отдельно не распознается компилятором C#, поэтому, строго говоря, ему не место в этом разделе книги. Однако же он определяет арифметические операторы и преобразования, что означает, что вы можете использовать его так же, как встроенные типы данных. Он будет скомпилирован в несколько менее компактный код, так как формат скомпилированных программ .NET может представлять целые числа и значения с плавающей точкой нативно, но **BigInteger** вынужден полагаться на более обобщенные механизмы, используемые обычными типами библиотек классов. В теории это будет происходить еще и значительно медленнее, хотя в огромном количестве кода скорость, с которой выполняется базовая арифметика для маленьких целых чисел, не является ограничивающим фактором. Вполне возможно, что вы даже не заметите никаких изменений. Что касается модели программирования, в коде **BigInteger** выглядит как обычный числовой тип.

Как следует из названия, **BigInteger** — это целочисленный тип. Его уникальным преимуществом является то, что он будет увеличиваться по мере роста своего значения. Таким образом, в отличие от встроенных числовых типов, теоретически у него нет границ диапазона. Листинг 2.39 использует его для вычисления значений в последовательности Фибоначчи, выводя каждое 100 000-е значение. Числа быстро становятся слишком большими, чтобы вместиться в любой из других целочисленных типов. Я показал полный исходный код этого примера, включая использование директив, чтобы проиллюстрировать, что этот тип определен в пространстве имен **System.Numerics**.

Хотя **BigInteger** не устанавливает фиксированных границ, существуют практические ограничения. Например, вы можете произвести слишком

большое число, чтобы оно поместилось в доступной памяти. Или, что более вероятно, числа могут вырасти настолько, что затраты процессорного времени, необходимого для выполнения даже базовой арифметики, станут непомерно большими. Но пока у вас хватает памяти и терпения, `BigInteger` будет расти, чтобы вмещать требуемые числа.

Листинг 2.39. Использование BigInteger

```
using System;
using System.Numerics;

class Program
{
    static void Main(string[] args)
    {
        BigInteger i1 = 1;
        BigInteger i2 = 2;
        Console.WriteLine(i1);
        int count = 0;
        while (true)
        {
            if (count++ % 100000 == 0)
            {
                Console.WriteLine(i2);
            }
            BigInteger next = i1 + i2;
            i1 = i2;
            i2 = next;
        }
    }
}
```

Логические типы

C# определяет тип `bool`, или, как его называет среда выполнения, `System.Boolean`. Он может содержать только два значения: `true` и `false`. В то время как некоторые языки семейства С допускают использование числовых типов для логических значений, соглашаясь, что 0 означает `false`, а все остальное — `true`, C# не примет число в качестве значения. Он требует, чтобы значения, обозначающие истинность или ложность, были представлены в виде `bool`, и ни один из числовых типов нельзя преобразовать в `bool`. Например, в операторе `if` вы не можете написать `if (some Number)`, чтобы заставить часть кода работать только тогда, когда `someNumber` не равно нулю.

Если это требуемое поведение, то следует явно указать на это, написав `if (someNumber != 0)`.

Строки и символы

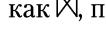
Тип `string` (синонимичный типу CLR `System.String`) представляет собой текст. Стока — это последовательность значений типа `char` (или `System.Char`, как его называет CLR), где каждый символ представляет собой 16-битное значение, являющееся одним кодовым квантом UTF-16.

Распространенная ошибка — думать, что каждый `char` представляет собой символ. (Часть вины за это лежит на названии типа.) Чаще всего это так, но далеко не всегда. Необходимо помнить о двух факторах: во-первых, то, что мы можем рассматривать как один символ, может состоять из нескольких кодовых позиций `Unicode`. (Кодовая позиция — ключевая концепция `Unicode`, и, по крайней мере в английском варианте, каждый символ представлен единственной кодовой позицией, но некоторые языки более сложные.) В листинге 2.40 используется код `0301`, называемый `COMBINING ACUTE ACCENT` для добавления знака ударения над буквой в слове `caf s`.

Листинг 2.40. Символ против `char`

```
char[] chars = { 'c', 'a', 'f', 'e', (char) 0x301, 's' };
string text = new string(chars);
```

Таким образом, строка представляет собой последовательность из шести значений символов, но создается впечатление, что текст содержит всего пять символов. Есть и другие способы добиться этого — я мог бы использовать кодовую позицию `00E9` "LATIN SMALL LETTER E WITH ACUTE", чтобы представить этот акцентированный символ единственной кодовой позицией. Но любой из этих подходов допустим, и существует множество сценариев, в которых единственный способ точно задать требуемый символ — это использовать механизм комбинирования символов. Это означает, что некоторые операции над значениями `char` в строке могут привести к неожиданным результатам — если обратить порядок значений, результирующая строка не будет выглядеть как развернутая версия исходного текста, так как акцент теперь будет расположен над `s`, в результате чего получится ` efac!` (Если бы я использовал `00E9` вместо комбинирования `e` с `0301`, обратный порядок символов привел бы к более ожидаемому `s fac`.)

Помимо присоединения меток к символам `Unicode` следует учесть еще один фактор. Стандарт `Unicode` определяет больше кодовых позиций, чем может быть представлено в одном 16-битном значении. (Мы прошли эту точку еще в 2001 году, когда `Unicode 3.1` определил 94 205 кодовых позиций.) `UTF-16` представляет любую кодовую позицию со значением выше 65 535 в виде пары кодовых единиц `UTF-16`, которая называется *суррогатной парой*. Стандарт `Unicode` определяет правила отображения кодовых позиций для суррогатных пар так, чтобы результирующие кодовые кванты содержали значения в диапазоне от `0xD800` до `0xFFFF`, что составляет зарезервированный диапазон, для которого никогда не будут определены кодовые позиции. (Например, кодовая позиция `10C48`, "OLD TURKIC LETTER ORKHON BASH", которая выглядит как , превратится в последовательность `0xD803` и `0xDC48`.)



Библиотека классов .NET содержит класс `StringInfo`, который способен помочь вам с задачей объединения символов. .NET Core 3.0 содержит новый тип `Rune` в пространстве имён `System`, который предоставляет различные вспомогательные методы, способные упростить работу с последовательностями из многокодовых квантов.

Подводя итог: элементы, которые пользователи воспринимают как отдельные символы, могут быть представлены несколькими кодовыми позициями `Unicode`, а некоторые отдельные кодовые позиции могут быть представлены как несколько кодовых квантов. Таким образом, манипулирование отдельными значениями `char`, составляющими `string`, следует выполнять с осторожностью.

Неизменность строк

Строки .NET являются неизменяемыми. Есть много операций, которые выглядят так, как будто они изменяют строку. В качестве примеров можно привести конкатенацию или методы `ToUpper` и `ToLower`, предлагаемые экземплярами `string`, но все они генерирует новую строку, оставляя исходную неизмененной. Это означает, что, даже если вы передадите строки в качестве аргументов в код, который писали не вы, можете быть уверены — он не сможет изменить эти строки.

Недостатком неизменяемости является то, что обработка строк может быть затратной. Если вам нужно выполнить задачу, включающую серию модификаций строки, например построить ее посимвольно, то в конечном

итоге на это уйдет много памяти, потому что для каждой модификации будет создаваться новая строка. Это создает много дополнительной работы для сборщика мусора .NET, заставляя вашу программу использовать больше процессорного времени, чем необходимо. В подобных ситуациях можно использовать тип `StringBuilder`. (В отличие от `string`, этот тип не обрабатывается компилятором C# особым образом.) Он концептуально похож на строку — это последовательность значений `char` плюс различные полезные методы манипуляции со строками, — но содержащаяся в нем строка модифицируема. В качестве альтернативы в чрезвычайно критичных в плане производительности сценариях вы можете использовать методы, о которых говорится в главе 18.

Форматирование данных в строках

C# поддерживает синтаксис, который позволяет легко создавать строки, содержащие смесь фиксированного текста и информации, определяемой во время выполнения. (Официальное название для этой функции — интерполяция строк.) Например, если у вас есть локальные переменные с именами и возрастом, вы можете использовать их в строке, как показано в листинге 2.41.

Листинг 2.41. Выражения в строках

```
string message = $"{name} is {age} years old";
```

Когда вы помещаете перед строковой константой символ \$, компилятор C# ищет встроенные выражения, разделенные фигурными скобками, и создает код, который вставит текстовое представление выражения в этой точке строки. (Таким образом, если бы `name` и `age` были `Ian` и `46` соответственно, значением строки было бы `"Ian is 46 years old"`.) Вложенные выражения могут быть более сложными, чем просто имена переменных, что показано в листинге 2.42.

Листинг 2.42. Более сложные выражения в строках

```
double width = 3, height = 4;
string info = $"Hypotenuse: {Math.Sqrt(width * width + height *
height)}";
```

Интерполированные строки компилируются в код, который использует метод `Format` класса `string` (именно так обычно выполнялось такое форматирование данных в более старых версиях C# — интерполяция строк была

введена в C# 6). Листинг 2.43 показывает код, который примерно соответствует тому, что будет генерировать компилятор для листингов 2.41 и 2.42.

Листинг 2.43. Эффект строковой интерполяции

```
string message = string.Format("{0} is {1} years old", name, age);
string info = string.Format(
    "Hypotenuse: {0}",
    Math.Sqrt(width * width + height * height));
```

Но почему бы просто не использовать базовый механизм `string.Format` напрямую? Строковая интерполяция намного менее подвержена ошибкам — `string.Format` использует позиционные заполнители, а так слишком легко поместить выражение в неправильное место. Любому, кто читает код, покажется утомительным выяснить, как пронумерованные заполнители соотносятся с последующими аргументами, особенно с ростом количества таких выражений. Интерполированные строки читать обычно намного легче.

Для некоторых типов данных можно выбирать вид их текстового представления. Например, для чисел с плавающей точкой вы можете захотеть ограничить количество десятичных знаков или принудительно использовать экспоненциальную запись (например, `1e6` вместо `1000000`). В .NET мы управляем этим с помощью спецификатора формата, который представляет собой строку, описывающую, как преобразовать некие данные в строку. Некоторые типы данных имеют только одно разумное строковое представление, поэтому они не поддерживают эту функцию, но что касается типов, которые это делают, то вы можете передать спецификатор формата в качестве аргумента методу `ToString`. Например, `System.Math.PI.ToString ("f4")` форматирует константу `PI` (которая имеет тип `double`) в число с четырьмя знаками после запятой ("3.1416"). Существует девять встроенных форматов для чисел, и если ни один из них не соответствует вашим требованиям, существует также мини-язык для задания пользовательских форматов. Более того, в разных типах используются строки разных форматов. Как и следовало бы ожидать, даты работают совершенно иначе, чем числа, поэтому полный диапазон доступных форматов слишком велик, чтобы можно было перечислить их все. По этой теме Microsoft предоставляет обширную и подробную документацию.

При использовании `string.Format` вы можете включить спецификатор формата в заполнитель; например, `{0: f3}` указывает, что первое выражение должно быть отформатировано в представление с тремя цифрами после десятичной точки. Аналогичным образом вы можете включить спецификатор

формата в интерполяцию строк. Листинг 2.44 показывает возраст с одной цифрой после десятичной точки.

Листинг 2.44. Спецификаторы формата

```
string message = $"{{name}} is {age:f1} years old";
```

Здесь кроется одна проблема: со многими типами данных процесс преобразования в строку зависит от конкретной культуры¹⁰. Например, как уже упоминалось ранее, в США и Великобритании десятичные дроби обычно пишутся с точкой между целочисленной и дробной частью, плюс вы можете использовать запятые для группировки цифр в целях удобства чтения. Но в некоторых европейских странах все наоборот: точки используются для группировки цифр, а запятая обозначает начало дробной части. Так что число, записываемое в одной стране как 1,000.2, может быть записано как 1.000,2 — в другой.

Что касается числовых литералов в исходном коде, это не проблема: C# использует подчеркивания для группировки цифр и всегда использует точку в качестве десятичной точки. Но что насчет форматирования чисел во время выполнения? По умолчанию вы подчиняетесь соглашениям, определяемым культурой текущего потока, так что если ничего не менять, то будут использоваться региональные настройки компьютера. Иногда полезно, что числа, даты и т. д. правильно отформатированы для любого региона, в котором работает программа. Тем не менее это может превратиться в проблему, если ваш код зависит от определенного формата строк (например, в случае сериализации данных, которые будут передаваться по сети). В этом случае вам может потребоваться ввести дополнительный набор определенных соглашений. По этой причине вы можете передавать методу `string.Format` создатель формата, объект, который управляет соглашениями о форматировании. Аналогично типы данных с культурно-зависимыми представлениями принимают создатель формата в качестве необязательного аргумента для своих методов `ToString`. Но как управлять всем этим при использовании интерполяции строк? Там некуда вставить создатель формата.

Эту проблему можно решить, поместив интерполированную строку в переменную типа `FormattableString` или `IFormattable` или же передав ее методу, который требует аргумент любого из этих типов¹¹. Сделав так, вы заставите

¹⁰ Под «культурой» здесь и далее подразумевается совокупность параметров, имеющих отношение к региональным особенностям продукта. — Примеч. ред.

¹¹ `IFormattable` — это интерфейс. Интерфейсы описаны в главе 3.

компилятор C# генерировать другой код: вместо непосредственного создания строки он создаст объект, который позволяет контролировать культурно-зависимое форматирование. Листинг 2.45 иллюстрирует эту технику на той же строке, что и в листинге 2.44.

Листинг 2.45. Спецификаторы формата с инвариантной культурой

```
string message = FormattableString.Invariant($"{name} is {age:f1}  
years old");
```

Тип `FormattableString` определяет два статических метода, `Invariant` и `CurrentCulture`, каждый из которых принимает аргумент типа `FormattableString`. Передав нашу интерполированную строку в один из них, мы заставим компилятор генерировать код, который оборачивает строку в `FormattableString`.

`FormattableString` реализует `IFormattable`, что дает дополнительный метод `ToString`, который принимает создатель формата, используемый для форматирования каждого из заполнителей в интерполированной строке. Используемый в листинге 2.45 метод `Invariant` вызывает этот метод, передавая создатель формата для инвариантной культуры. Этот создатель (который также можно получить из свойства `CultureInfo.InvariantCulture`) гарантирует согласованное форматирование независимо от региона, в котором выполняется код. Если вы вызываете `FormattableString.CurrentCulture`, он отформатирует строку с учетом текущей культуры потока.

Буквальные строковые литералы

C# поддерживает еще один способ выражения строкового значения: вы можете предварить строковый литерал префиксом @, например @"Hello". Строки такого вида называются буквальными строковыми литералами. Они удобны по двум причинам: они способны улучшить читаемость строк, содержащих обратную косую черту, а также позволяют записывать многострочные строковые литералы.



Символ @ можно использовать перед интерполированной строкой. Это позволит совместно использовать преимущества дословных литералов — прямое использование обратной косой черты и новых строк — и поддержку встроенных выражений.

В случае обычного строкового литерала компилятор обрабатывает обратную косую черту как экранирующий символ, позволяя включать в нее различные

специальные значения. Например, в литерале "Hello \tworlд" \t обозначает один символ табуляции (кодовая позиция 9). Это распространенный способ добавления управляющих символов в языках семейства С. Вы также можете использовать обратную косую черту для включения в строку двойных кавычек — обратная косая черта не позволяет компилятору интерпретировать символ как конец строки. Хотя это и полезно, но саму косую черту добавлять в строку не очень удобно: ее нужно написать дважды. Так как Windows использует обратную косую черту в путях, результат может выглядеть довольно неприглядно, например C:\\Windows\\System32\\. Здесь может быть полезен буквальный строковый литерал, потому что он обрабатывает обратную косую черту буквально и позволяя записать подобную строку просто как @"C:\\Windows\\System32". (Вы все равно можете включить двойные кавычки в дословный литерал, напечатав две двойные кавычки подряд: например, @"Hello""world"" даст строковое значение Hello "World".)

Буквальные строковые литералы также позволяют значениям занимать несколько строк. С обычным строковым литералом компилятор сообщит об ошибке, если закрывающая двойная кавычка не находится в той же строке, что и открывающая. Но с буквальным строковым литералом строка может занимать столько строк исходного кода, сколько вам нужно.

Полученная строка будет следовать любому соглашению о конце строки, использующемуся в вашем исходном коде. На тот случай, если вы с этим никогда не сталкивались, одна из злополучных трудностей в истории вычислений состоит в том, что разные системы используют разные последовательности символов для обозначения концов строк. Преобладающей системой в интернет-протоколах является использование пары контрольных кодов для каждого конца строки: в Unicode или ASCII мы используем кодовые точки 13 и 10, обозначающие возврат каретки и перевод строки соответственно, часто сокращаемые до CR LF. Это архаичное наследие времен, когда у компьютеров была растровая сетка и начало новой строки означало перемещение печатающей головки телетайпа обратно в исходное положение (возврат каретки), а затем перемещение бумаги на одну строку вверх (перевод строки). Аналогичные спецификация HTTP требует именно этого представления, как и различные популярные стандарты электронной почты, такие как SMTP, POP3 и IMAP. Так же это является стандартным соглашением в Windows. К сожалению, операционная система Unix работает по-другому, как и большинство ее производных и аналогичных программ, включая macOS и Linux. В этих системах принято использовать только один символ перевода строки. Компилятор C# не будет жаловаться, даже если один исходный файл

следует сразу обоим соглашениям. Это создает потенциальную проблему в случае многострочных строковых литералов, когда вы используете систему контроля версий, которая преобразует окончания строк за вас. Например, git является очень популярной системой управления исходным кодом, и благодаря ее корням (она была сделана Линусом Торвальдсом, создателем Linux) в ее репозиториях широко используется соглашение о конце строк в стиле Unix. Однако в Windows ее можно настроить для преобразования рабочих копий файлов в представление CR LF, автоматически преобразовывая их обратно в LF при принятии изменений. Это означает, что файлы будут выглядеть по-разному в зависимости от того, в какой системе вы их открываете, в Windows или Unix. (Это может меняться и от одной системы Windows к другой, потому что обработку окончания строки по умолчанию можно изменить. Отдельные пользователи могут настроить параметры по умолчанию для всей системы, а также настроить конфигурацию для своего локального клона любого репозитория, если репозиторий сам не устанавливает этот параметр.) Это, в свою очередь, означает, что компиляция в Windows файла, содержащего многострочный дословный строковый литерал, может вести себя немного иначе, чем при использовании того же файла в системе Unix, при условии, что включено автоматическое преобразование конца строки (что по умолчанию используется в большинстве Windows-установок git). Это может быть и хорошо — вы обычно желаете видеть CR LF при работе в Windows и LF в Unix, — но может привести к неожиданностям, если развернуть код на машине, работающей под управлением другой ОС, чем та, на которой вы его создали. Поэтому важно иметь в своих репозиториях файл `.gitattributes`, чтобы иметь возможность указывать требуемое поведение, не полагаясь на изменяемые локальные настройки. Если для вас важен определенный тип окончания строк в строковых литералах, лучше сделать так, чтобы ваши `.gitattributes` отключали преобразования конца строки.

Кортежи

В C# 7.0 появилась новая языковая функция: поддержка кортежей, которые позволяют объединять несколько значений в одно. Имя `tuple` (которое используется в C# вместе со многими другими языками программирования, предоставляющими аналогичную функцию) представляет собой обобщенную версию таких слов, как `double`, `triple`, `quadruple` и т. д. Мы обычно называем их кортежами (`tuples`) даже в тех случаях, когда нам не требуется это обобщение. Например, когда речь идет о кортеже с двумя элементами,

мы все равно называем его `tuple`, а не `double`. Листинг 2.46 создает кортеж, содержащий два значения типа `int`, и затем отображает их.

Листинг 2.46. Создание и использование кортежа

```
(int X, int Y) point10 = (5, 10);
Console.WriteLine($"X: {point.X}, Y: {point.Y}");
```

Первая строка — это объявление переменной с блоком инициализации. Это стоит отметить отдельно, потому что синтаксис кортежей делает объявление несколько более сложным, чем мы видели до сих пор. Вспомним, что общий шаблон для подобных операторов выглядит так:

идентификатор типа = начальное значение;

Это означает, что в листинге 2.46 типом является `(int X, int Y)`. Итак, мы сообщаем, что наша переменная `point` — это кортеж, содержащий два значения типа `int`, на которые мы хотим ссылаться как на `X` и `Y`. Блоком инициализации здесь служит `(10, 5)`. Поэтому когда мы запустим пример, то получим такой результат:

```
X: 10, Y: 5
```

Если вы фанат `var`, вам будет приятно узнать, что можно указывать имена в инициализаторе, используя синтаксис, показанный в листинге 2.47, что позволит вам использовать `var` вместо явного типа. Это эквивалентно листингу 2.46.

Листинг 2.47. Именование членов кортежа в блоке инициализации

```
var point = (X: 10, Y: 5);
Console.WriteLine($"X: {point.X}, Y: {point.Y}");
```

Если вы инициализируете кортеж из существующих переменных без указания имен, компилятор предполагает, что вы хотите использовать имена этих переменных, как показано в листинге 2.48.

Листинг 2.48. Получение имен членов кортежа из переменных

```
int x = 10, y = 5;
var point = (x, y);
Console.WriteLine($"X: {point.x}, Y: {point.y}");
```

Это поднимает стилистический вопрос: со строчных или прописных букв следует начинаться именам членов кортежа? Члены по своей природе по-

хожи на свойства, которые мы обсудим в главе 3, и, как правило, начинаются с заглавной буквы. По этой причине многие считают, что имена членов кортежа также должны начинаться с заглавной буквы. В глазах опытного разработчика .NET `point.x` из листинга 2.48 выглядит странновато. Однако другое соглашение .NET заключается в том, что имена локальных переменных обычно начинаются со строчной буквы. Если вы придерживаетесь обоих этих соглашений, наследование имени кортежа выглядит не очень подходящей практикой. Многие разработчики допускают имена членов кортежа в нижнем регистре для кортежей, используемых исключительно в локальных переменных, потому что это позволяет использовать удобную функцию наследования имен, а конвенцию Pascal используют только для кортежей, которые доступны вне метода.

В принципе, это не имеет большого значения, потому что имена членов кортежа существуют только в глазах смотрящего. Во-первых, они не обязательны. Как показано в листинге 2.49, их вполне можно опустить и использовать имена по умолчанию: `Item1`, `Item2` и т. д.

Листинг 2.49. Имена членов кортежа по умолчанию

```
(int, int) point = (10, 5);
Console.WriteLine($"X: {point.Item1}, Y: {point.Item2}");
```

Во-вторых, имена используются исключительно для удобства чтения кода и не видны во время выполнения. Вы заметите, что я использовал то же выражение для блока инициализации `(10, 5)`, что и в листинге 2.46. Поскольку он не определяет имена, выражение выглядит как `(int, int)`, что соответствует листингу 2.49. Но я также смог назначить их прямо в `(int X, int Y)` в листинге 2.46. Это потому, что имена по сути не имеют отношения к делу — под капотом все это выглядит совершенно одинаково. (Как мы увидим в главе 4, во время выполнения все они представлены как экземпляры типа `ValueTuple<int, int>`.) Компилятор C# отслеживает имена, которые мы выбрали, но что касается CLR, то для него все эти кортежи просто содержат члены с именами `Item1` и `Item2`. В результате мы можем присвоить любой кортеж любой переменной той же формы, как показано в листинге 2.50.

Листинг 2.50. Структурная эквивалентность кортежей

```
(int X, int Y) point46 = (3, 46);
(int Width, int Height) dimensions = point;
(int Age, int NumberOfChildren) person = point;
```

Эта гибкость — обоюдоострый меч. Присвоения в листинге 2.50 выглядят довольно схематично. Предположительно можно присвоить нечто представляющее местоположение чему-то, что представляет размер, — и в некоторых ситуациях это вполне допустимо. Но присваивать это же значение чему-то, что представляет возраст и количество детей, скорее всего, неправильно. Компилятор не остановит нас, потому что для него все кортежи, содержащие пару значений `int`, имеют одинаковый тип. (На самом деле это ничем не отличается от того факта, что компилятор не пресечет назначение переменной `int` с именем `age` в переменную `int` с именем `height`. Оба они типа `int`.)

Если вам нужно учитывать семантическое различие, лучше определить собственные типы, как это описано в главе 3. В действительности кортежи разработаны как удобный способ объединения нескольких значений в случаях, когда определение целого нового типа не оправдано.

Само собой, C# требует, чтобы кортежи имели соответствующую форму. Вы не сможете присвоить `(int, int)` ни `(int, string)`, ни `(int, int, int)`. Однако все неявные преобразования, описанные в «Числовых преобразованиях» на с. 92, сработают, поэтому вы сможете присвоить все, что имеет форму `(int, int)`, чему-то вроде `(int, double)` или `(double, long)`. Таким образом, кортеж на самом деле напоминает несколько переменных, ловко втиснутых в другую переменную.

Кортежи поддерживают сравнение, поэтому вы можете использовать операторы сравнения `==` и `!=`, описанные далее в этой главе. Чтобы считаться равными, два кортежа должны иметь одинаковую форму, и каждое значение в первом кортеже должно быть равно его двойнику во втором.

Деконструкция

Иногда вы захотите разделить кортеж обратно на составные части. Очевидно, что можно просто друг за другом обратиться к каждому элементу по имени (или же `Item1`, `Item2` и т. д., если вы не указали имена), но в C# имеется другой механизм, называемый деконструкцией. Листинг 2.51 объявляет и инициализирует два кортежа, после чего демонстрирует два разных способа их деконструкции.

Определив `point1` и `point2`, пример разбивает `point1` на две переменные, `x` и `y`. Эта конкретная форма деконструкции также объявляет переменные, в которые кортеж деконструируется. Другой способ демонстрируется на примере деконструкции `point2` — здесь мы разбиваем кортеж на две пере-

менные, которые уже существуют, поэтому нет необходимости их дополнительно объявлять.

Листинг 2.51. Построение и деконструкция кортежей

```
(int X, int Y) point1 = (40, 6);
(int X, int Y) point2 = (12, 34);

(int x, int y) = point1;
Console.WriteLine($"1: {x}, {y}");
(x, y) = point2;
Console.WriteLine($"2: {x}, {y}");
```

Пока вы не привыкнете к этому синтаксису, первый пример деконструкции может казаться странно похожим на первую пару строк, в которых мы объявляем и инициализируем новые кортежи. В этих первых двух строках текст `(int X, int Y)` обозначает тип кортежа с двумя значениями `int` и именами `X` и `Y`, но в строке деконструкции, когда мы пишем `(int x, int y)`, мы фактически объявляем две локальные переменные типа `int`. Единственное существенное отличие состоит в том, что в строках, где мы создаем новые кортежи, перед знаком `=` стоит имя переменной. (Кроме того, мы используем там заглавные буквы, но это просто вопрос соглашения. Вполне допустимым было бы написать `(int x, int y) point3 = point1;` Это объявило бы новый кортеж с двумя значениями `int` с именами `x` и `y`, которые хранятся в переменной с именем `point3`, а инициализируются значениями, которые содержатся в `point1`. Точно так же мы могли бы написать `(int X, int Y) = point1;` Это разобьет `point` на две локальные переменные, с именами `X` и `Y`.)

Тип `dynamic`

C# определяет тип, называемый `dynamic`. Напрямую он не соответствует ни одному типу CLR — когда мы используем `dynamic` в C#, компилятор представляет его среди выполнения как объект, который описывается в следующем разделе. Тем не менее с точки зрения кода C# `dynamic` — это отдельный тип, обладающий некоторыми особенностями поведения.

При использовании типа `dynamic` компилятор не пытается во время компиляции проверить вероятность того, будут ли успешными выполняемые кодом операции. Иными словами, он по сути отключает обусловленное статической типизацией поведение, которое мы обычно ожидаем от C#. С динамической переменной можно попытаться выполнить практически любую операцию: использовать арифметические операторы, вызвать ее методы, присвоить ее

переменным другого типа, установить или получить ее свойства. Когда вы делаете что-то подобное, компилятор генерирует код, с помощью которого можно наиболее разумными методами добиться желаемого результата.

Если вы перешли на C# с языка, в котором такое поведение является нормой (скажем, JavaScript), у вас может возникнуть соблазн использовать `dynamic` для всего подряд, потому что он работает так, как вы привыкли. Тогда вам стоит знать о паре его слабых мест. Во-первых, он разработан с учетом конкретного сценария использования, а именно совместимости с некоторыми компонентами Windows, существовавшими до .NET. Модель компонентного объекта (COM) в Windows лежит в основе автоматизированности Microsoft Office Suite и множества других приложений, а сценарный язык, встроенный в Office, динамический по своей природе. Результатом стало то, что многие API автоматизации Office крайне затруднительно использовать из C#. Желание преодолеть это затруднение послужило одним из главных факторов, которые привели к добавлению в язык динамического типа. Как и все остальные функциональные средства C#, он был разработан с учетом более широкого применения, а не просто как способ взаимодействия с Office. Но, поскольку именно этот сценарий использования был поставлен во главу угла, вас может разочаровать уровень поддержки типом `dynamic` ряда идиом, знакомых вам по другим языкам. Есть и вторая проблема, о которой следует знать. Динамическая типизация — это не та область языка, над которой ведется работа на постоянной основе. Представляя эту функцию, Microsoft сделала все, чтобы динамическое поведение максимально соответствовало поведению, которое можно ожидать от кода в случае, когда компилятор знает, какие типы данных вы собираетесь использовать. Это означает, что инфраструктура поддержки типа `dynamic` (называемая `Dynamic Language Runtime`, или DLR) вынуждена во многом воспроизводить поведение C#. Несмотря на множество последующих нововведений языка, DLR практически не обновлялся с тех самых пор, когда динамический тип был представлен в C# 4.0 в далеком 2010 году. Несомненно, `dynamic` еще работает, но его возможности отражают то, каким язык был около десяти лет назад.

Даже впервые появившись, он уже имел некоторые ограничения. Ряд аспектов C# зависят от наличия статической информации о типе, из-за чего использование `dynamic` всегда было чревато проблемами при работе с делегатами, а также с LINQ. Таким образом, уже с самого начала тип `dynamic` был своего рода узким местом по сравнению с надлежащим использованием C#, т. е. как статически типизированного языка.

Тип `object`

Последний тип данных, который особым образом распознается компилятором C#, — это `object` (или `System.Object`, как его называет CLR). Это базовый класс почти для всех типов C#¹². Переменная типа `object` может ссылаться на значение любого типа, производного от `object`. Это включает в себя все числовые типы, типы `bool` и `string`, а также любые пользовательские типы, которые вы можете определить с помощью ключевых слов, которые мы рассмотрим в следующей главе, таких как `class` и `struct`. Кроме того, он включает в себя все типы, определенные библиотекой классов .NET, за исключением определенных типов, которые могут храниться только в стеке и которые описаны в главе 18.

Таким образом, `object` — это обобщенный контейнер общего назначения. С помощью переменной типа `object` вы можете ссылаться практически на что угодно. Мы вернемся к этому в главе 6, когда будем говорить о наследовании.

Операторы

Ранее вы уже видели, что выражения — это последовательности операторов и операндов. Я продемонстрировал некоторые типы, которые можно использовать в качестве операндов, так что теперь пришло время посмотреть, какие же операторы предлагает C#. В табл. 2.3 показаны операторы, которые производят элементарные арифметические операции.

Если вы знакомы с любым другим языком семейства C, все это должно показаться вам знакомым. Если нет, то, вероятно, наиболее специфичными покажутся операторы инкремента и декремента. У них есть побочные эффекты: они добавляют или вычтут единицу из переменной, к которой применяются (т. е. они могут применяться только к переменным). В случае постинкремента и постдекремента, хотя переменная и изменяется, содержащее выражение в итоге получает исходное значение. Таким образом, если `x` — это переменная, содержащая значение 5, то значение `x++` также равно 5, даже если переменная `x` будет иметь значение 6 после вычисления выражения `x++`. Преинкремент и преддекремент возвращают измененное значение, поэтому, если `x` изначально равно 5, `++x` вернет 6, что также является значением `x` после вычисления выражения.

¹² Есть несколько особых исключений, таких как типы указателей.

Таблица 2.3. Основные арифметические операторы

Название	Пример
Унарный плюс (ничего не делает)	<code>+x</code>
Отрицание (унарный минус)	<code>-x</code>
Постинкремент	<code>x++</code>
Постдекремент	<code>x--</code>
Преинкремент	<code>++x</code>
Преддекремент	<code>--x</code>
Сложение	<code>x + y</code>
Вычитание	<code>x - y</code>
Умножение	<code>x * y</code>
Деление	<code>x / y</code>
Остаток	<code>x % y</code>

Хотя операторы из табл. 2.3 используются в арифметике, некоторые доступны для ряда нечисловых типов. Как вы уже видели ранее, символ `+` производит конкатенацию при работе со строками, и, как вы еще увидите в главе 9, операторы сложения и вычитания также используются для объединения и удаления делегатов.

C# также предлагает операторы, которые выполняют определенные двоичные операции над битами, составляющими значение переменной. Они перечислены в табл. 2.4. Они недоступны для типов с плавающей точкой.

Таблица 2.4. Бинарные целочисленные операторы

Название	Пример
Побитовое отрицание	<code>~x</code>
Побитовое AND	<code>x & y</code>
Побитовое OR	<code>x y</code>
Побитовое XOR	<code>x ^ y</code>
Сдвиг влево	<code>x << y</code>
Сдвиг вправо	<code>x >> y</code>

Оператор побитового отрицания инвертирует все биты в целом числе — любой двоичный символ со значением 1 становится 0, и наоборот. Операторы сдвига перемещают все двоичные символы влево или вправо на количество

столбцов, указанное вторым операндом. Сдвиг влево устанавливает нижние цифры в 0. Сдвиги вправо целых чисел без знака заполняют верхние цифры значением 0, а сдвиги вправо целых чисел со знаком оставляют верхний символ нетронутым (т. е. отрицательные числа остаются отрицательными, поскольку они сохраняют свой верхний бит установленным, в то время как положительные числа сохраняют свой верхний бит как 0, оставаясь положительными).

Битовые операторы **AND**, **OR** и **XOR** (исключающее ИЛИ) в применении к целым числам выполняют булевые логические операции с каждым битом двух operandов. Эти три оператора работают и когда operandы имеют тип **bool**. (По сути, эти операторы обрабатывают **bool** как однозначное двоичное число.) Есть еще несколько дополнительных операторов, доступных для **bool** и показанных в табл. 2.5. Оператор **!** делает с **bool** то же, что **~** делает с каждым битом в целом числе.

Таблица 2.5. Операторы для **bool**

Название	Пример
Логическое отрицание (также известное как NOT)	<code>!x</code>
Условное AND	<code>x&& y</code>
Условное OR	<code>x y</code>

Если вы не использовали другие языки семейства C, условные версии операторов **AND** и **OR** могут оказаться для вас новыми. Они вычисляют свой второй operand только при необходимости. Например, при оценке (**a && b**), если выражение **a** представляет собой **false**, код, сгенерированный компилятором, не будет пытаться вычислить **b**, потому что результат будет ложным независимо от значения **b**. И наоборот, оператор условного **OR** не удосуживается вычислять свой второй operand, если первый равен **true**, потому что результат будет истинным независимо от значения второго operandана. Это важно, если выражение второго operandана имеет побочные эффекты (например, оно включает вызов метода) или способно вызвать ошибку. Например, можно часто встретить код, подобный приведенному в листинге 2.52.

Листинг 2.52. Оператор условного AND

```
if (s != null && s.Length > 10)
...
...
```

Код проверяет, содержит ли переменная `s` специальное значение `null`, означающее, что в данный момент она не ссылается ни на какое значение. Здесь важно использовать оператор `&&`, потому что если `s` равно `null`, вычисление выражения `s.Length` вызовет ошибку времени выполнения. Если бы мы использовали оператор `&`, компилятор сгенерировал бы код, который всегда оценивает оба операнда, что означает, что мы увидим исключение `NullReferenceException` во время выполнения, если `s` равно `null`; однако, используя условный оператор `AND`, мы избегаем этого, поскольку второй operand, `s.Length > 10`, будет оцениваться только в том случае, если `s` не равен `null`.



Хотя код того типа, что показан в листинге 2.52, когда-то был обычным делом, он постепенно уступает позиции благодаря функции, введенной еще в C# 6.0, а именно `null`-условным операторам. Если вы напишете `s?.Length` вместо просто `s.Length`, компилятор генерирует код, который сначала проверит `s` на `null`, что позволит избежать исключения `NullReferenceException`. Это означает, что проверка может превратиться просто в `if (s?.Length > 10)`. Кроме того, в C# 8.0 появилась новая функция, согласно которой вы можете указать, что определенные значения никогда не должны быть `null`, что может помочь уменьшить потребность в таких типах проверок на `null`. Это обсуждается в главе 3.

В листинге 2.52 с помощью оператора `>` проверяется, превышает ли свойство значение 10. Это один из нескольких операторов сравнения, которые позволяют сравнивать значения. Все они требуют два операнда и выдают результат типа `bool`. Они перечислены в табл. 2.6 и поддерживаются для всех числовых типов. Некоторые операторы доступны и для ряда других типов. Например, вы можете сравнить строковые значения с помощью операторов `==` и `!=`. (Не существует встроенного механизма работы со строками для других операторов сравнения, потому что разные страны имеют разные представления о порядке сортировки строк. Если вы хотите сравнить упорядоченные строки, .NET предлагает класс `StringComparer`, который потребует от вас выбрать правила, по которым вы хотите упорядочивать ваши строки.)

Как это обычно принято в языках семейства C, оператор равенства представляет собой пару знаков равенства. Это потому, что один знак равенства означает нечто другое, а именно присваивание. А присваивания тоже являются выражениями. Это приводит к известной проблеме: в некоторых языках семейства C можно легко написать `if (x = y)`, имея в виду `if (x == y)`.

К счастью, в C# это обычно приводит к ошибке компилятора, потому что в C# имеется специальный тип для представления логических значений. В языках, которые позволяют заменять логические значения числами, оба фрагмента кода допустимы, даже если `x` и `y` являются числами. (Первый означает присвоение значения переменной `y` переменной `x`, после чего выполняется тело оператора `if`, если полученное значение отлично от нуля. Это поведение сильно отличается от второго, где код не меняет значение чего-либо и выполняет тело оператора `if`, только если `x` и `y` равны.) Но в C# первый пример будет иметь смысл, только если тип обеих переменных `bool`¹³.

Таблица 2.6. Операторы сравнения

Название	Пример
Меньше, чем	<code>x < y</code>
Больше, чем	<code>x > y</code>
Меньше или равно	<code>x <= y</code>
Больше или равно	<code>x >= y</code>
Равно	<code>x == y</code>
Не равно	<code>x != y</code>

Другая особенность, которая является общей для семейства C, — это условный оператор. (Его иногда называют троичным оператором, потому что это единственный оператор в языке, который принимает три операнда.) Он осуществляет выбор между двумя выражениями. Точнее, он оценивает свой первый operand, который должен быть логическим выражением, а затем возвращает значение второго или третьего операнда в зависимости от того, было ли значение первого истинным или ложным. Листинг 2.53 использует его, чтобы выбрать большее из двух значений. (Этот пример служит только для иллюстрации. На практике вы обычно используете метод `.NET Math.Max`, который имеет тот же эффект, но воспринимается гораздо легче. `Math.Max` имеет еще и то преимущество, что если используются выражения с побочными эффектами, он будет вычислять каждое из них только один раз, чего нельзя сказать о подходе, показанном в листинге 2.53, потому что в результате мы написали каждое выражение дважды.)

¹³ Буквоеды заметят, что это также будет иметь смысл в определенных ситуациях, когда доступны пользовательские неявные преобразования в `bool`. Мы обратимся к пользовательским преобразованиям в главе 3.

Листинг 2.53. Условный оператор

```
int max = (x > y) ? x : y;
```

Это показывает, почему С и его потомки имеют репутацию языков с лаконичным синтаксисом. Если вы знакомы с любым языком из этого семейства, листинг 2.53 будет легко читаемым, но если нет, его смысл может быть понятен не сразу. Здесь происходит вычисление выражения перед символом ?, которое в данном случае выглядит как (x > y), а его результатом должен стать bool. (Скобки необязательны. Я добавил их, чтобы облегчить чтение кода.) Если результат true, то используется выражение между символами ? и : (в данном случае x); в противном случае используется выражение после символа : (в данном случае y).

Условный оператор аналогичен условным операторам AND и OR в том, что он будет вычислять только те операнды, которые необходимо. Он всегда вычисляет первый operand, но никогда не будет вычислять второй или третий. Это означает, что вы можете обрабатывать значения null, написав что-то вроде листинга 2.54. Здесь нет риска вызвать исключение NullReferenceException, потому что оператор будет вычислять третий operand, только если s не null.

Листинг 2.54. Использование условного вычисления

```
int characterCount = s == null ? 0 : s.Length;
```

Однако в некоторых случаях есть более простые способы работы с null. Предположим, у вас есть строковая переменная, и если она равна null, вы хотите использовать вместо нее пустую строку. Вы могли бы написать (s == null? "" : s). Но вместо этого вы можете просто использовать оператор объединения с null, потому что он именно для этого и предназначен. Этот оператор, показанный в листинге 2.55 (символ ??), вычисляет свой первый operand, и, если он не равен null, делает его результатом выражения. Если первый operand все же null, он вычисляет второй и использует его.

Листинг 2.55. Оператор объединения с null

```
string neverNull = s ?? "";
```

Можно объединить null-условный оператор с оператором объединения с null, что даст более лаконичную альтернативу листинга 2.54, что показано в листинге 2.56.

Листинг 2.56. Null-условный оператор и оператор объединения с null

```
int characterCount = s?.Length ?? 0;
```

Одно из основных преимуществ условных, `null`-условных и операторов объединения с `null` заключается в том, что они зачастую позволяют вам написать единственное выражение, когда в ином случае вам понадобился бы значительно больший объем кода. Это особенно полезно, если в качестве аргумента метода вы используете выражение, как демонстрируется в листинге 2.57.

Листинг 2.57. Условное выражение как аргумент метода

```
FadeVolume(gateOpen ? MaxVolume : 0.0, FadeDuration, FadeCurve.Linear);
```

Подумайте, сколько кода вам пришлось бы написать в случае отсутствия условного оператора. Вам бы понадобился оператор `if`. (Я вернусь к операторам `if` в следующем разделе, но, поскольку эта книга не для новичков, я предполагаю, что в целом идея вам понятна.) И вам нужно будет либо ввести локальную переменную, как в листинге 2.58, или же дублировать вызов метода в двух ветвях `if/else`, изменения только первый аргумент. Таким образом, несмотря на лаконичность условных операторов и операторов слияния с `null`, они способны во многом избавить вас от беспорядка в коде.

Листинг 2.58. Жизнь без условного оператора

```
double targetVolume;
if (gateOpen)
{
    targetVolume = MaxVolume;
}
else
{
    targetVolume = 0.0;
}
FadeVolume(targetVolume, FadeDuration, FadeCurve.Linear);
```

Нам осталось рассмотреть последний набор операторов: составные операторы присваивания. Они объединяют присваивание с некоторыми другими операциями, такими как `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, `|` и `??`. Они позволяют вам избежать написания кода, показанного в листинге 2.59.

Листинг 2.59. Присваивание и сложение

```
x = x + 1;
```

Мы можем сделать этот оператор присваивания более компактным, как в листинге 2.60. Все составные операторы присваивания принимают подобную форму — вы просто добавляете символ `=` в конце исходного оператора.

Листинг 2.60. Составное присваивание (сложение)

```
x += 1;
```

Это особый синтаксис, который очевидным образом показывает, что мы изменяем значение переменной конкретным способом. Таким образом, хотя эти два фрагмента и выполняют одинаковую функцию, многие разработчики предпочитают второй вариант.

Это далеко не полный список операторов. Есть еще несколько специализированных, к которым я вернусь, когда мы перейдем к областям языка, для которых они предназначены. (Некоторые относятся к классам и другим типам, некоторые — к наследованию, некоторые — к коллекциям, а некоторые — к делегатам. Всем этим вопросам посвящены отдельные главы.) Кстати, хотя я описывал, какие операторы доступны для каких типов (например, числовые или логические), можно создать собственный тип, который определяет собственный смысл для большинства из них. Вот почему тип `.NET BigInteger` поддерживает те же арифметические операции, что и встроенные числовые типы. Я покажу, как это можно сделать, в главе 3.

Управление потоком

Большая часть кода, который мы до сих пор рассматривали, выполняет операторы в порядке их написания и останавливается, когда достигает конца. Если бы это было единственным возможным способом выполнения нашего кода, C# был бы не очень полезен. Но, как и следовало ожидать, в нем имеется множество конструкций для написания циклов и принятия решений на основе входных данных о том, какой код выполнять.

Выбор на основе булевой логики в операторе if

Оператор `if` решает, следует ли выполнять некий конкретный оператор, в зависимости от значения `bool` какого-либо выражения. Например, оператор `if` в листинге 2.61 выполнит блок, который показывает сообщение, только если значение переменной `age` меньше 18.

Листинг 2.61. Простой оператор if

```
if (age < 18)
{
    Console.WriteLine("You are too young to buy alcohol in a bar in
                      the UK.");
}
```

Вам не обязательно использовать с оператором `if` оператор блока. Вы можете вместо этого использовать любой тип оператора. Блок необходим лишь в том случае, если вы хотите, чтобы оператор `if` управлял выполнением нескольких операторов. Однако некоторые рекомендации касательно стиля кодирования предписывают во всех случаях использовать блок. Отчасти в целях согласованности, но также и потому, что это позволяет избежать возможных ошибок при изменении кода на более позднем этапе: если у вас в качестве тела оператора `if` имеется оператор вне блока, а вы затем добавляете еще один оператор после него, намереваясь сделать его частью того же тела, можно легко забыть заключить их в один блок, что приведет к появлению кода, подобного показанному в листинге 2.62. Отступ предполагает, что разработчик имел в виду, что последний оператор входит в тело оператора `if`, но C# игнорирует отступы, так что конечный оператор будет выполняться всегда. Если у вас есть привычка всегда использовать блок, вы не совершиете эту ошибку.

Листинг 2.62. Вероятно, все было задумано не так

```
if (launchCodesCorrect)
    TurnOnMissileLaunchedIndicator();
    LaunchMissiles();
```

Оператор `if` может дополнительно включать в себя часть `else`, за которой следует другой оператор, который выполняется, только если выражение оператора `if` ложно. Поэтому листинг 2.63 будет выводить либо первое, либо второе сообщение в зависимости от содержимого переменной `optimistic`, которое может быть `true` или `false`.

Листинг 2.63. `If` и `else`

```
if (optimistic)
{
    Console.WriteLine("Glass half full");
}
else
{
    Console.WriteLine("Glass half empty");
}
```

За ключевым словом `else` может следовать любой оператор, и, опять же, обычно это блок. Однако есть один сценарий, в котором большинство разработчиков не используют блок для тела `else`, а именно когда они используют еще один оператор `if`. Листинг 2.64 показывает это — его первый

оператор `if` имеет часть `else`, в качестве тела которой используется еще один оператор `if`.

Листинг 2.64. Выбор одной из нескольких возможностей

```
if (temperatureInCelsius < 52)
{
    Console.WriteLine("Too cold");
}
else if (temperatureInCelsius > 58)
{
    Console.WriteLine("Too hot");
}
else
{
    Console.WriteLine("Just right");
}
```

Этот код все еще выглядит так, как будто он использует блок для первого `else`, но этот блок на самом деле является оператором, который составляет тело второго оператора `if`. Именно этот второй оператор `if` является телом `else`. Если бы мы строго придерживались правила предоставления каждому условию `if` и `else` собственного блока, мы переписали бы листинг 2.64, получив листинг 2.65. Это выглядит излишним, потому что основной риск, который мы пытаемся предотвратить с помощью блоков, на самом деле никогда не сработает в листинге 2.64.

Листинг 2.65. Перебор с блоками

```
if (temperatureInCelsius < 52)
{
    Console.WriteLine("Too cold");
}
else
{
    if (temperatureInCelsius > 58)
    {
        Console.WriteLine("Too hot");
    }
    else
    {
        Console.WriteLine("Just right");
    }
}
```

Хотя мы можем объединять операторы `if`, как показано в листинге 2.64, C# предлагает специализированный оператор, как правило более читаемый.

Множественный выбор с оператором `switch`

Оператор `switch` определяет несколько групп операторов и либо запускает одну группу, либо ничего не делает в зависимости от значения входного выражения. Как показано в листинге 2.66, после ключевого слова `switch` вы помещаете в круглые скобки выражение, после чего следует область, ограниченная фигурными скобками и содержащая серию разделов `case`, определяющих поведение для каждого из ожидаемых значений выражения.

Листинг 2.66. Оператор `switch` со строками

```
switch (workStatus)
{
    case "ManagerInRoom":
        WorkDiligently();
        break;

    case "HaveNonUrgentDeadline":
    case "HaveImminentDeadline":
        CheckTwitter();
        CheckEmail();
        CheckTwitter();
        ContemplateGettingOnWithSomeWork();
        CheckTwitter();
        CheckTwitter();
        break;

    case "DeadlineOvershot":
        WorkFuriously();
        break;

    default:
        CheckTwitter();
        CheckEmail();
        break;
}
```

Как видите, один раздел может обслуживать несколько вариантов — вы можете поместить в начале раздела несколько разных меток `case`, и операторы в этом разделе будут выполняться, если хотя бы одна из них является результатом вычисления выражения соответствия. Вы также можете до-

бавить раздел по умолчанию, который будет выполняться, если ни один из остальных разделов не сработал. Кстати, добавлять его совсем не обязательно. Оператор `switch` не обязан быть исчерпывающим, поэтому, если нет ни одного `case`, соответствующего значению выражения, как нет и раздела по умолчанию, оператор `switch` просто ничего не делает.

В отличие от операторов `if`, которые принимают для тела только один оператор, за `case` могут следовать несколько операторов без необходимости заключать их в блок. Разделы в листинге 2.66 ограничены операторами `break`, что заставляет выполнение переходить к концу оператора `switch`. Это не единственный способ завершить раздел — строго говоря, компилятор C# устанавливает правило, согласно которому конечная точка списка операторов для каждого случая не должна быть достижимой, поэтому все, что вызывает выполнение инструкции `switch`, приемлемо. Вы можете использовать оператор `return`, выдать исключение или даже использовать оператор `goto`.

Некоторые языки семейства С (например, сам С) допускают сквозное выполнение, что означает, что если выполнение достигает конца раздела `case`, оно переходит к следующему. Листинг 2.67 демонстрирует этот стиль, но он не разрешен в C# из-за правила, которое требует, чтобы конец списка операторов `case` был недостижимым.

Листинг 2.67. Сквозное выполнение в стиле С, недопустимое в C#

```
switch (x)
{
    case "One":
        Console.WriteLine("One");
    case "Two": // Эта строка не скомпилируется
        Console.WriteLine("One or two");
        break;
}
```

C# запрещает это, потому что подавляющее большинство разделов `case` не предназначены для сквозного выполнения. Когда в допускающих сквозное выполнение языках происходит нечто подобное, чаще всего это ошибка, вызванная тем, что разработчик забыл написать оператор `break` (или какой-либо другой оператор, ведущий к выходу из `switch`). Случайное сквозное выполнение может привести к нежелательному поведению, поэтому в C# для сквозного выполнения потребуется нечто большее, чем простой пропуск `break`: если вам нужно такое поведение, его нужно явно запросить. В листин-

где 2.68 используется всеми нелюбимое ключевое слово `goto`, чтобы показать, что мы действительно хотим, чтобы один `case` перешел в следующий.

Листинг 2.68. Сквозное выполнение в C#

```
switch (x)
{
    case "One":
        Console.WriteLine("One");
        goto case "Two";
    case "Two":
        Console.WriteLine("One or two");
        break;
}
```

Технически это не оператор `goto`, а оператор `goto case`, который можно использовать только для перехода в блок `switch`. C# поддерживает и более распространенные операторы `goto`, когда вы можете добавлять в код метки и перемещаться внутри своих методов. Как бы то ни было, к оператору `goto` многие относятся неодобрительно, так что сквозное выполнение с помощью оператора `goto case`, похоже, осталось единственным применением этого ключевого слова, снискавшим одобрение общественности.

Во всех этих примерах используются строки. Но вы также можете использовать `switch` с целочисленными типами, `char` и любым `enum` (тип, обсуждаемый в следующей главе). В течение многих лет это были практически все варианты, потому что метки `case` должны были быть константами. Но C# 7.0 добавил в оператор `switch` поддержку шаблонов в метках `case`. Шаблоны обсуждаются далее в этой главе.

Циклы `while` и `do`

C# поддерживает обычные для семейства языков С разновидности цикла. В листинге 2.69 показан цикл `while`. Он принимает выражение `bool`, вычисляет его и, если результат равен `true`, выполняет последующий оператор. Это напоминает оператор `if`, но разница в том, что, как только встроенный в цикл оператор завершается, цикл снова вычисляет выражение, и, если оно по-прежнему `true`, выполняет встроенный оператор во второй раз. Это продолжается до тех пор, пока выражение не становится `false`. Как и в случае с операторами `if`, тело цикла не обязательно должно быть блоком, но обычно это так.

Листинг 2.69. Цикл `while`

```
while (!reader.EndOfStream)
{
    Console.WriteLine(reader.ReadLine());
}
```

Тело цикла может решить завершить цикл раньше времени с помощью оператора `break`. В этом случае не важно, является ли выражение `while` истинным или ложным — выполнение оператора `break`, безусловно, завершает цикл.

В C# также есть оператор `continue`. Как и оператор `break`, он завершает текущую итерацию, но, в отличие от `break`, после этого он заново вычисляет выражение `while`, поэтому итерации могут продолжаться. И `continue`, и `break` переносят выполнение в конец цикла, но вы можете думать о `continue` как о прыжке к непосредственно закрывающей тело цикла фигурной скобке `}`, в то время как `break` ведет к прыжку в точку сразу после нее. Кстати, `continue` и `break` также доступны для всех других разновидностей цикла, о которых я собираюсь рассказать.

Поскольку оператор `while` вычисляет свое выражение перед каждой итерацией, цикл `while` может вообще ни разу не исполнить свое тело. Иногда вам может понадобиться написать цикл, который выполняется хотя бы один раз, вычисляя булево выражение только после первой итерации. Так работает цикл `do`, что показано в листинге 2.70.

Листинг 2.70. Цикл `do`

```
char k;
do
{
    Console.WriteLine("Press x to exit");
    k = Console.ReadKey().KeyChar;
}
while (k != 'x');
```

Обратите внимание, что в конце кода листинга 2.70 стоит точка с запятой, обозначающая конец оператора. Сравните это со строкой, содержащей ключевое слово `while` в листинге 2.69, в которой этого нет, хотя в остальном она выглядит очень похоже. Это может выглядеть противоречием, но это не опечатка. Поместить точку с запятой в конец строки с ключевым словом `while` в листинге 2.69 было бы допустимо, но это изменило бы весь смысл. Это указало бы на то, что мы хотим, чтобы тело цикла `while` было пустым оператором. Последующий же блок будет рассматриваться как совершенно

новый оператор, выполняемый после завершения цикла. Код застрял бы в бесконечном цикле, если только `reader` уже не был бы в конце потока. (Кстати, если вы это сделаете, компилятор выдаст предупреждение «Possible mistaken empty statement».)

Циклы в стиле C

Другой стиль цикла, который C# унаследовал от C, – это цикл `for`. Он похож на `while`, но добавляет две функции к выражению условия: предоставляет место для объявления и/или инициализации одной или нескольких переменных, которые будут оставаться в области действия до тех пор, пока выполняется цикл, а также предоставляет место для выполнения некоторой операции один раз за цикл (в дополнение к оператору, который формирует тело цикла). Таким образом, структура цикла `for` выглядит так:

```
for (инициализация; условие; итератор) тело
```

Очень распространенное применение такого цикла состоит в том, чтобы проделать что-то с каждым элементом массива. Листинг 2.71 показывает цикл `for`, который умножает каждый элемент в массиве на 2. Часть, касающаяся условия, работает точно так же, как в цикле `while`, – определяет, выполняется ли встроенный оператор, формирующий тело цикла, и вычисляется перед каждой итерацией. Опять же, тело не обязательно должно быть блоком, но обычно это так.

Листинг 2.71. Изменение элементов массива с помощью цикла for

```
for (int i = 0; i < myArray.Length; i++)
{
    myArray[i] *= 2;
}
```

Блок инициализации в этом примере объявляет переменную с именем `i` и присваивает ей значение `0`. Конечно, эта инициализация происходит только один раз, так как было бы не очень удобно, если бы она во время выполнения цикла каждый раз сбрасывала переменную на `0`, потому что тогда бы цикл никогда не закончился. Время жизни этой переменной по сути начинается непосредственно перед началом цикла и заканчивается с окончанием цикла. Блок инициализации не обязательно должен быть объявлением переменной – можно использовать любой оператор выражения.

Итератор в листинге 2.71 просто добавляет `1` к счетчику цикла. Он запускается в конце каждой итерации цикла, после выполнения тела и до пересчета

условия. (Таким образом, если условие изначально ложно, тело не только не запускается, но и итератор никогда не будет вычислен.) С# ничего не делает с результатом итератора — он полезен только за счет побочного использования. Поэтому не имеет значения, пишете вы `i++, ++i, i += 1` или даже `i = i + 1`.

Цикл `for` не позволяет делать ничего, чего нельзя достичь с помощью цикла `while`, поместив код инициализации перед циклом, а итератор — в конец тела цикла. Тем не менее код может выиграть за счет удобства чтения¹⁴. Оператор `for` помещает код, определяющий, как мы выполняем цикл, отдельно от кода, который определяет, что мы делаем вокруг цикла, что может помочь читающим код понять его назначение. Им не нужно просматривать длинный цикл до конца, чтобы найти оператор итератора (хотя тело цикла, которое продолжается на протяжении нескольких страниц кода, обычно считается плохой практикой, поэтому последнее преимущество несколько сомнительно).

И инициализатор, и итератор могут содержать списки, как показано в листинге 2.72, хотя в данном конкретном случае это не очень полезно — из-за того, что все итераторы запускаются каждый раз, `i` и `j` будут иметь одинаковое значение на протяжении всего времени.

Листинг 2.72. Несколько инициализаторов и итераторов

```
for (int i = 0, j = 0; i < myArray.Length; i++, j++)
...
```

Вы не можете написать единственный цикл `for`, который выполняет многомерную итерацию. При такой необходимости вложите один цикл в другой, как показано в листинге 2.73.

Листинг 2.73. Вложенные циклы for

```
for (int j = 0; j < height; ++j)
{
    for (int i = 0; i < width; ++i)
    {
        ...
    }
}
```

¹⁴ Оператор `continue` все усложняет, поскольку дает возможность перейти к следующей итерации, не доходя до конца тела цикла. Тем не менее вы можете воспроизвести эффект итератора и при использовании операторов `continue`, просто для этого потребуется больше работы.

Хотя в листинге 2.71 показана достаточно распространенная идиома для прохода по массивам, вы чаще будете использовать другую, более специализированную конструкцию.

Итерация по коллекции с использованием цикла `foreach`

C# предлагает разновидность цикла, которая не является общепринятой в языках семейства С. Цикл `foreach` предназначен для итерирования коллекций и соответствует следующей схеме:

```
foreach (item-type iteration-variable in collection) body
```

Коллекция — это выражение, тип которого должен соответствовать определенному шаблону, распознаваемому компилятором. Интерфейс библиотеки классов .NET `IEnumerable<T>`, который мы рассмотрим в главе 5, соответствует этому шаблону, хотя компилятору на самом деле не требуется реализация этого интерфейса — ему достаточно, чтобы коллекция содержала метод `GetEnumerator`, который напоминает метод, определенный этим интерфейсом. Листинг 2.74 использует `foreach`, чтобы вывести все строки в массиве. (Все массивы содержат метод, который требует `foreach`.)

Листинг 2.74. Перебор коллекции с помощью `foreach`

```
string[] messages = GetMessagesFromSomewhere();
foreach (string message in messages)
{
    Console.WriteLine(message);
}
```

Этот цикл выполняет тело один раз для каждого элемента в массиве. Итерационная переменная (в данном примере `message`) отличается каждый проход цикла, ссылаясь при этом на элемент для текущей итерации.

С одной стороны, это поведение менее гибкое, чем у цикла на основе `for`, показанного в листинге 2.71: цикл `foreach` не способен модифицировать коллекцию, по которой он проходит. Это потому, что не все коллекции поддерживают модификацию. `IEnumerable<T>` требует от коллекции очень немногое: ему не требуется модифицируемость, произвольный доступ или даже возможность заранее знать, сколько элементов содержит коллекция. (На самом деле `IEnumerable<T>` способен поддерживать бесконечные коллекции. Например, совершенно допустимо написать реализацию, которая

будет возвращать случайные числа до тех пор, пока вы продолжаете извлекать значения.)

Но `foreach` имеет перед `for` два преимущества. Одно из них является субъективным и поэтому спорным: цикл немного более удобен для чтения. Но что более важно, он является и более всеобъемлющим. Если вы пишете методы, которые работают с коллекциями, они будут более широко применимы, если в них использовать `foreach` вместо `for`, потому что тогда вы сможете принимать `IEnumerable<T>`. Листинг 2.75 может работать с любой коллекцией, содержащей строки, не ограничиваясь лишь массивами.

Листинг 2.75. Обобщенная итерация по коллекции

```
public static void ShowMessages(IEnumerable<string> messages)
{
    foreach (string message in messages)
    {
        Console.WriteLine(message);
    }
}
```

Этот код может работать с типами коллекций, которые не поддерживают произвольный доступ, такими как класс `LinkedList<T>` (см. главу 5). Еще он может обрабатывать отложенные (ленивые) коллекции, которые определяют, какие элементы создавать по требованию, в том числе с помощью функций итератора, что тоже показано в главе 5, а также определенных запросов LINQ, как описано в главе 10.

Шаблоны

В C# есть еще один ключевой механизм: шаблоны. Шаблон описывает один или несколько критериев, по которым можно проверить значение. Вы уже видели несколько простых шаблонов в действии: каждый `case` в `switch` определяет шаблон. Но, как мы сейчас увидим, существует много видов шаблонов, предназначенных не только для операторов `switch`.



Большая часть функционала шаблонов была добавлена в C# относительно недавно. Поддержка их впервые появилась в C# 7.0, но большинство доступных типов шаблонов были добавлены в C# 8.0.

Предыдущие примеры использования `switch`, такие как листинг 2.66, содержали один из самых простых типов шаблонов: шаблон константы. С помощью этого шаблона вы задаете только постоянное значение, и выражение соответствует этому шаблону, если оно имеет указанное значение. Аналогичным образом в листинге 2.76 показаны шаблоны кортежа, которые сравнивают кортежи с конкретными значениями. Концептуально они очень похожи на шаблоны константы, так как в них значения для отдельных элементов кортежа являются константами. (Различие между шаблонами константы и шаблонами кортежа во многом является историческим: до того как шаблоны были введены, метки `case` поддерживали только ограниченный набор типов, для которых CLR предлагает встроенную поддержку постоянных значений, и этот список не включает кортежи.)

Листинг 2.76. Шаблоны кортежа

```
switch (p)
{
    case (0, 0):
        Console.WriteLine("How original");
        break;

    case (0, 0):
    case (1, 1):
        Console.WriteLine("What an absolute unit");
        break;

    case (1, 1):
        Console.WriteLine("Be there and be square");
        break;
}
```

В листинге 2.77 показан более интересный тип шаблона, а именно *шаблон типа*. Выражение соответствует шаблону типа, если оно указанного типа. Как вы видели ранее в разделе «Тип `object`» на с. 114, некоторые переменные могут содержать различные типы. Переменные типа `object` доводят этот принцип до крайности, поскольку могут содержать с некоторыми оговорками все, что угодно. Языковые функции, такие как *интерфейсы* (см. главу 3), обобщения (глава 4) и наследование (глава 6), могут привести к сценариям, когда статический тип переменной дает больше информации, чем тип `object`, в который можно поместить что угодно. При этом остается некоторая свобода выбора возможных типов во время выполнения. В этом случае могут пригодиться шаблоны типа.

Листинг 2.77. Шаблоны типа

```
switch (o)
{
    case string s:
        Console.WriteLine($"A piece of string is {s.Length} long");
        break;

    case int i:
        Console.WriteLine($"That's numberwang! {i}");
        break;
}
```

Шаблоны типа имеют интересную особенность, которой нет у шаблонов константы: кроме логической проверки на совпадение, общей для всех шаблонов, шаблон типа производит еще и дополнительный вывод. Каждый `case` в листинге 2.77 создает переменную, которая затем используется в этом же `case`. То, что мы видим в качестве вывода, — это просто входные данные, скопированные в переменную с указанным статическим типом. Таким образом, первый `case` сработает, если `o` окажется строкой, и в этом случае мы сможем получить к ней доступ через переменную `s` (вот почему выражение `s.Length` компилируется; строка `o.Length` вызовет ошибку компиляции, если `o` имеет тип `object`).



Вам не всегда нужен вывод шаблона типа, так как может быть достаточно просто знать, что входные данные соответствуют шаблону. В этих случаях вы можете использовать механизм сброса: если вписать подчеркивание (`_`) туда, где должно быть имя выходной переменной, это скажет компилятору C#, что вас интересует лишь соответствие значения типу.

Некоторые шаблоны производят немного больше работы для получения своего вывода. Например, листинг 2.78 демонстрирует шаблон позиции, который проверяется на соответствие любому кортежу из пары значений `int`, после чего эти значения извлекаются в переменные `x` и `y`.

Листинг 2.78. Шаблон позиции

```
case (int x, int y):
    Console.WriteLine($"I know where it's at: {x}, {y}");
    break;
```

Шаблоны позиции служат примером рекурсивного шаблона: это шаблоны, которые содержат в себе шаблоны. В данном случае шаблон позиции включает в себя шаблон типа как каждый из своих дочерних элементов. Это дает большое разнообразие, потому что шаблоны позиции могут содержать любой типа шаблона (включая еще один рекурсивный шаблон, если это потребуется). Фактически именно это происходило в листинге 2.76, где шаблон кортежа — это на самом деле просто особый случай шаблона позиции, где все дочерние элементы являются шаблонами константы. В листинге 2.79 показан шаблон позиции с шаблоном константы в первой позиции и шаблоном типа во второй.

Листинг 2.79. Шаблон позиции с шаблонами константы и типа

```
case (0, int y):
    Console.WriteLine($"This is on the X axis, at height {y}");
    break;
```

Если вы фанат `var`, то поинтересуетесь, можно ли написать что-то вроде листинга 2.80. Можно, и статические типы переменных `x` и `y` здесь будут зависеть от типа входного выражения шаблона. Если компилятор способен определить, как деконструировать выражение (например, если статический тип входных данных оператора `switch` является кортежем (`int, int`)), он будет использовать эту информацию для определения статических типов выходных переменных. В случаях, когда это неизвестно, но соответствие все же возможно (например, вход представляет собой `object`), тогда `x` и `y` в данном случае также будут иметь тип `object`.

Листинг 2.80. Шаблон позиции с `var`

```
case (var x, var y):
    Console.WriteLine($"I know where it's at: {x}, {y}");
    break;
```



Компилятор отклоняет шаблоны в тех случаях, когда может точно определить, что совпадение невозможно. Например, если известно, что тип входных данных представляет собой кортеж (`string, int, bool`), то его нет смысла сравнивать с шаблоном позиции всего с двумя дочерними элементами, так что C# не даст вам даже попробовать.

В листинге 2.80 показан необычный случай, когда использование `var` вместо явного типа может привести к заметному изменению поведения. *Шаблоны*

переменной отличаются от *шаблонов типа* из листинга 2.78 в одном важном аспекте: *шаблон переменной* всегда соответствует своим входным данным, тогда как *шаблон типа* проверяет тип, чтобы определить соответствие во время выполнения. На практике эта проверка может быть оптимизирована — бывают случаи, когда шаблон типа всегда будет соответствовать, так как тип его входных данных известен во время компиляции. Но единственный способ показать в своем коде, что вы точно не хотите, чтобы дочерние шаблоны в шаблоне позиции выполняли проверку во время выполнения, — это использовать `var`. Таким образом, хотя позиционный шаблон, содержащий шаблоны типа, сильно напоминает синтаксис деконструкции, показанный в листинге 2.51, поведение у него совершенно иное. Код из листинга 2.78 во время выполнения в действительности производит три проверки: является ли значение кортежем из двух элементов, является ли первое значение `int`, является ли второе `int?` (Таким образом, он будет работать для кортежей со статическим типом (`object`, `object`), если каждое значение является `int` во время выполнения.) Это не должно вызывать удивления: смысл шаблонов в том, чтобы во время выполнения проверять, соответствует ли значение определенным характеристикам. Однако в случае ряда рекурсивных шаблонов вам может понадобиться комбинация проверки на соответствие во время выполнения (например, является ли строка `string?`) и статически типизированной деконструкции (например, если это `string`, то мне необходимо извлечь ее свойство `Length`, которое, по моему убеждению, должно быть типа `int`, и я хочу получить ошибку компилятора, если это убеждение окажется неверным). Шаблоны не предназначены для подобных задач, поэтому лучше не пытаться использовать их таким образом.

Но что, если не требуется использовать все элементы кортежа? Один способ вам уже известен. Поскольку в каждой позиции мы можем использовать любой шаблон, то нам доступен и шаблон типа со сбросом, скажем, во второй позиции: `(int x, int _)`. Тем не менее в листинге 2.81 показана более лаконичная альтернатива: вместо сброса в шаблоне типа мы можем использовать единственный символ подчеркивания. Это *шаблон сброса*. Вы можете использовать его в рекурсивном шаблоне везде, где требуется шаблон, в том числе и там, где в конкретной позиции подойдет что угодно, а вам не нужно знать, что именно там оказалось.

Листинг 2.81. Шаблон позиции с шаблоном сброса

```
case (int x, _):
    Console.WriteLine($"At X: {x}. As for Y, who knows?");
    break;
```

Тем не менее у него несколько иная семантика: шаблон типа собросом во время выполнения будет проверять, имеет ли сбрасываемое значение заданный тип, и шаблон в целом пройдет проверку только в случае успешной проверки типа. Но шаблонброса проходит проверку всегда, поэтому он будет соответствовать, например, (10, 20), (10, "Foo") и (10, (20, 30)).

Позиционные шаблоны – не единственные рекурсивные шаблоны. Кроме них вы также можете написать шаблон свойства. Мы подробно рассмотрим свойства в следующей главе, но на данный момент достаточно знать, что они являются членами типа, содержащими некоторую информацию, например свойство `Length` типа `string`, имеющее тип `int`, указывает на количество кодовых единиц в строке. В листинге 2.82 показан шаблон свойства, который проверяет то самое свойство `Length`.

Листинг 2.82. Шаблон свойства

```
case string { Length: 0 }:  
    Console.WriteLine("How long is a piece of string? Not very!");  
    break;
```

Шаблон свойства начинается с имени типа, поэтому он по сути включает в себя поведение шаблона типа в дополнение к собственной проверке на основе свойств. (Вы можете опустить это в тех случаях, когда тип входных данных шаблона достаточно специфичен для определения свойства. Например, это можно опустить, если входные данные в данном случае уже имели статический тип `string`.) После этого следует раздел в фигурных скобках, в котором перечисляются все свойства, которые шаблону требуется проверить, и шаблон, который нужно применить к свойству. (Эти дочерние шаблоны и делают его еще одним рекурсивным шаблоном.) Таким образом, в первую очередь пример проверяет, является ли входная строка `string`. Если это так, то он применяет шаблон константы к длине строки, поэтому совпадение будет только в том случае, если входные данные представляют собой `string`, где `Length` равно 0.

Шаблоны свойства могутoptionально задавать выходные данные. Листинг 2.82 этого не делает. Листинг 2.83 показывает синтаксис, хотя в данном конкретном случае от него мало пользы, поскольку шаблон лишь гарантирует, что `s` всегда ссылается только на пустую строку.

Поскольку каждое свойство в шаблоне свойства содержит вложенный шаблон, они также могут создавать выходные данные, как показано в листинге 2.84.

Листинг 2.83. Шаблон свойства с выводом

```
case string { Length: 0 } s:  
    Console.WriteLine($"How long is a piece of string? This long:  
                      {s.Length}");  
    break;
```

Листинг 2.84. Шаблон свойства с вложенным шаблоном с выводом

```
case string { Length: int length }:  
    Console.WriteLine($"How long is a piece of string? This long:  
                      {length}");  
    break;
```

Уточнение с помощью when

Иногда встроенные типы шаблонов не способны обеспечить требуемый уровень точности. Например, на примере шаблонов позиции мы поняли, как писать шаблоны, которые соответствуют, скажем, любой паре значений, или любой паре чисел, или паре чисел с определенным значением. Но что, если нужна проверка на совпадение с парой чисел, где первое больше второго? Концептуально отличие невелико, но встроенной поддержки для этого уже нет. Конечно, мы могли бы проверить условие с помощью оператора `if`, но было бы досадно переводить наш код со `switch` на серию операторов `if` и `else` лишь для того, чтобы сделать такой маленький шаг. К счастью, нам и не придется.

Любой шаблон в метке `case` можно уточнить, добавив раздел `when`, который позволяет использовать логическое выражение. Оно будет вычисляться, если значение соответствует основной части шаблона, но в целом значение будет соответствовать шаблону, только если раздел `when` возвращает `true`. В листинге 2.85 показан шаблон позиции с разделом `when`, который проверяет, что в паре чисел первое число больше второго.

Листинг 2.85. Шаблон с разделом `when`

```
case (int w, int h) when w > h:  
    Console.WriteLine("Landscape");  
    break;
```

Шаблоны в выражениях

Все шаблоны, которые я до сих пор демонстрировал, располагались в метках `case` оператора `switch`. Но это далеко не единственный способ использова-

ния шаблонов. Они также могут располагаться и внутри выражений. Чтобы увидеть, как это можно использовать, сначала взгляните на оператор `switch` в листинге 2.86. Его задача состоит в том, чтобы вернуть единственное значение, определенное входными данными, но выглядит это немного неуклюже: мне пришлось написать четыре отдельных оператора `return`.

Листинг 2.86. Шаблоны, но не в выражениях

```
switch (shape)
{
    case (int w, int h) when w < h: return "Portrait";
    case (int w, int h) when w > h: return "Landscape";
    case (int _, int _): return "Square";
    default: return "Unknown";
}
```

В листинге 2.87 показан код, который выполняет ту же работу, но переписан с помощью выражения `switch`. Как и в случае оператора `switch`, выражение `switch` содержит список шаблонов. Разница заключается в том, что если за метками в операторе `switch` следует список операторов, то в выражении `switch` каждый шаблон сопровождается одним выражением. Значение выражения `switch` является результатом вычисления выражения, связанного с первым совпадающим шаблоном.

Выражения `switch` выглядят совершенно иначе, чем операторы `switch`, потому что в них не используется ключевое слово `case`. Вместо этого они срабатывают прямо в шаблоне, где используются `=>` между шаблоном и его соответствующим выражением. Для этого есть ряд причин. Во-первых, это делает выражения `switch` немного более компактными. Выражения обычно используются внутри чего-то еще; например, в нашем случае выражение `switch` является значением оператора `return`. Но вы также можете использовать их как аргумент метода или где-либо еще, где разрешено использование выражения. Именно поэтому от них требуется быть лаконичными. Во-вторых, использование здесь `case` могло бы привести к путанице, потому что правила для того, что следует за каждым `case`, были бы разными для операторов `switch` и выражений `switch`: в операторе `switch` каждая метка `case` сопровождается одним или несколькими операторами, а в выражении `switch` каждый шаблон должен сопровождаться одним выражением. Наконец, хотя выражения `switch` были добавлены только в версии C# 8.0, такая конструкция встречается в других языках уже много лет. Его версия на C# гораздо больше похожа на аналоги из других языков, чем если бы в выражении `switch` использовалось ключевое слово `case`.

Листинг 2.87. Выражение switch

```
return shape switch
{
    (int w, int h) when w < h => "Portrait",
    (int w, int h) when w > h => "Landscape",
    (int _, int _) => "Square",
    _ => "Unknown"
};
```

Обратите внимание, что последний шаблон в листинге 2.87 представляет собой шаблон сброса. Он будет соответствовать чему угодно, и это гарантирует, что шаблон является исчерпывающим, т. е. охватывает все возможные случаи. (Он действует аналогично разделу по умолчанию в операторе `switch`.) В отличие от оператора `switch`, где вполне нормально, если совпадений нет, выражение `switch` должно выдавать результат, поэтому компилятор предупредит вас, если ваши шаблоны не обрабатывают все возможные варианты типов входных данных. В данной ситуации компилятор показал бы ошибку, если бы мы удалили последний `case`, предполагая, что входные данные `shape` имеют тип `object`. (И напротив, если бы `shape` имел тип `(int, int)`, нам следовало бы удалить последний `case`, потому что первые три случая фактически охватывают все возможные значения для этого типа и компилятор выдаст ошибку, сообщающую нам, что последний шаблон никогда не будет применен.) Если вы проигнорируете это предупреждение, а затем во время выполнения вычислите выражение `switch` с несопоставимым значением, будет выброшено исключение `SwitchExpressionException`. Исключения описаны в главе 8.

Есть еще один способ использовать шаблон в выражении — ключевое слово `is`. Оно превращает любой шаблон в логическое выражение. Листинг 2.88 наглядно определяет, является ли значение кортежем, содержащим два целых числа.

Листинг 2.88. Выражение is

```
bool isPoint = value is (int x, int y);
```

Как и в случае шаблонов в операторах или выражениях `switch`, шаблон в выражении `is` способен извлекать значения на основе своего исходного кода. В листинге 2.89 используется то же выражение, что и в предыдущем примере, но два значения из кортежа при этом используются в дальнейшем.

Новые переменные, подобным образом вводимые выражением `is`, остаются в области действия и после содержащего их оператора. Таким образом, в обоих примерах `x` и `y` останутся в области действия до конца содержащего

блока. Поскольку шаблон в листинге 2.89 находится в выражении условия оператора `if`, переменные остаются в области видимости после блока тела `if`. Но если вы попытаетесь использовать их вне тела, вы обнаружите, что, согласно определенным правилам присвоения, компилятор сообщат вам, что они не инициализированы. Он допускает листинг 2.89, потому что знает, что тело оператора `if` будет выполняться только при совпадении шаблона, поэтому `x` и `y` будут инициализированы и безопасны для использования.

Листинг 2.89. Использование значений из шаблона выражения `is`

```
if (value is (int x, int y))
{
    Console.WriteLine($"X: {x}, Y: {y}");
}
```

Шаблоны в выражениях `is` не могут включать раздел `when`. Это было бы избыточно: результатом и так является логическое выражение, так что вы можете просто добавить любое нужное уточнение, используя обычные логические операторы, как показано в листинге 2.90.

Листинг 2.90. В шаблоне выражения `is` нет нужды в шаблоне `when`

```
if (value is (int w, int h) && w < h)
{
    Console.WriteLine($"(Portrait) Width: {w}, Height: {h}");
}
```

Итог

В этой главе я показал основные моменты C#: переменные, операторы, выражения, основные типы данных, управление потоком и шаблоны. Теперь пришло время шире взглянуть на структуру программы. Весь код в программах на C# должен принадлежать какому-то типу, а типы являются темой следующей главы.

ГЛАВА 3

Типы

C# не ограничивает нас встроенными типами данных, показанными в главе 2. Вы можете определить свои собственные типы. Фактически у вас нет выбора: если вы вообще хотите писать код, C# требует, чтобы вы определили тип, который будет содержать этот код. Все, что мы напишем, а также любые используемые функции библиотеки классов .NET (или любой другой библиотеки .NET), будут принадлежать какому-либо типу.

C# распознает несколько разновидностей типов. Начну с самого важного.

Классы

Большинство типов, с которыми вы работаете в C#, — это *классы*. Класс может содержать как код, так и данные, а также может делать некоторые свои функции публичными, оставляя другие доступными только для кода внутри класса. Таким образом, классы предлагают механизм инкапсуляции — они могут определять понятный публичный программный интерфейс для использования другими людьми, сохраняя при этом внутренние детали реализации недоступными.

Если вы знакомы с объектно-ориентированными языками, все это покажется очень простым. Если нет, то, возможно, лучше сначала почитать книгу начального уровня, поскольку моя книга не предназначена для обучения программированию. Я просто опишу детали, специфичные для классов C#.

Я уже показал примеры классов в предыдущих главах, но давайте рассмотрим их структуру более подробно. В листинге 3.1 показан простой класс. (См. врезку «Условные обозначения» на с. 143 для получения информации об именах для типов и их членов.)

Определения классов всегда содержат ключевое слово `class`, за которым следует имя класса. C# не требует, чтобы имя соответствовало содержащемуся файлу, и при этом не ограничивает вас наличием одного класса в файле. Тем

не менее в большинстве проектов C# класс и имя файла совпадают. В любом случае в именах классов следует придерживаться основных правил для описанных в главе 2 идентификаторов, таких как переменные; например, они не могут начинаться с цифры.

Листинг 3.1. Простой класс

```
public class Counter
{
    private int _count;
    public int GetNextValue()
    {
        _count += 1;
        return _count;
    }
}
```

Первая строка листинга 3.1 содержит дополнительное ключевое слово: `public`. Определения класса могут дополнительно задавать доступность, которая определяет, как другой код может использовать данный класс. Обычные классы имеют только два варианта: `public` и `internal`, причем последний используется по умолчанию. (Как я покажу позже, классы можно вкладывать в другие типы, а вложенные классы имеют немного более широкий диапазон опций доступности.) Внутренний класс доступен для использования только в компоненте, который его определяет. Поэтому, если вы пишете библиотеку классов, вы можете определить классы, которые существуют исключительно как часть реализации вашей библиотеки: помечая их как внутренние, вы не позволяете использовать их извне.



Вы можете сделать свои внутренние типы видимыми для избранных внешних компонентов. Microsoft иногда проделывает это со своими библиотеками. Библиотека классов .NET распределена по многочисленным DLL, каждая из которых определяет множество внутренних типов, но некоторые внутренние функции используются другими DLL в составе библиотеки. Это стало возможным благодаря аннотированию компонентов с помощью атрибута `[assembly: InternalsVisibleTo("name")]`, указывающего имя компонента, с которым вы хотите поделиться. (Глава 14 описывает это более подробно.) Например, вам может потребоваться сделать каждый класс в приложении видимым для проекта, чтобы иметь возможность писать юнит-тесты для кода, который вы не собираетесь делать публичным.

СОГЛАШЕНИЯ ОБ ИМЕНАХ

Microsoft определяет ряд соглашений для публично видимых идентификаторов, которым она (в основном) следует в своих библиотеках классов, и я тоже обычно следую им в своих примерах. Microsoft предоставляет бесплатный анализатор FxCop, который может помочь в реализации этих соглашений. Вы можете подключить его для любого проекта, добавив ссылку на пакет NuGet `Microsoft.CodeAnalysis.FxCopAnalyzers`. Если вы просто хотите почитать описание этих правил, то они являются частью рекомендаций по проектированию библиотек классов .NET по адресу <https://docs.microsoft.com/dotnet/standard/design-guidelines/index>.

В этих соглашениях первая буква имени класса пишется с заглавной буквы, и если имя содержит несколько слов, каждое новое слово также начинается с заглавной буквы. (По историческим причинам это соглашение называется `Pascal Casing` или иногда `PascalCasing` в качестве автореферентного примера.) Хотя для идентификаторов C# допустимы подчеркивания, означенные соглашения не допускают их в именах классов. Методы и свойства также используют `Pascal casing`. Поля редко бывают открытыми, но когда они все же открыты, то для них используют одни и те же соглашения о регистре.

Параметры метода используют другое соглашение, известное как `camelCasing`, при котором заглавные буквы используются в начале всех слов, кроме первого. Он назван так потому, что это соглашение приводит к появлению одного или нескольких горбов в середине слова.

Рекомендации по проектированию библиотеки классов хранят молчание относительно деталей реализации. (Первоначальная цель этих правил и инструмента FxCop состояла в том, чтобы обеспечить единство во всем публичном API библиотеки классов .NET Framework. «Fx» — это сокращение от Framework.) Таким образом, эти правила ничего не говорят о том, как называть закрытые поля. Я использовал префикс подчеркивания в листинге 3.1, потому что мне нравится, когда поля выглядят иначе, чем локальные переменные. Так проще увидеть, с какими данными работает мой код, кроме того, это помогает избежать ситуаций, когда имена параметров методов конфликтуют с именами полей. (Microsoft использует это же соглашение для полей экземпляра в .NET Core вместе с префиксами `s_` и `t_` для статических полей и полей, локальных для потоков.) Некоторые люди считают это соглашение уродливым и предпочитают не делать визуальных различий между полями, но предпочитают всегда иметь доступ к членам с помощью ссылки `this` (описанной позже), так что различие между доступом к переменной и полем сохраняется.

Класс `Counter` в листинге 3.1 определен открытым, но это не значит, что все в нем должно быть публичным. Он определяет два члена — поле с именем

`_count`, содержащее `int`, и метод с именем `GetNextValue`, который работает с информацией в этом поле. (CLR при создании счетчика автоматически инициализирует это поле значением `0`.) Как видите, оба этих члена также имеют квалификаторы доступа. Как это часто бывает в объектно ориентированном программировании, элемент класса сделан закрытым, а открытая функциональность предоставляется через метод.

Модификаторы доступа являются необязательными для членов классов, так же как и для самих классов, и, опять же, они по умолчанию используют наиболее ограничивающий параметр: в данном случае `private`. Так что я мог бы без потерь опустить ключевое слово `private` в листинге 3.1, но я предполагаю сохранять код более наглядным. (Если его указывать, люди, читающие ваш код, могут задаться вопросом, было ли это упущение намеренным или случайным.)

Поля содержат данные. Они являются своего рода переменными, но в отличие от локальной переменной, область и время жизни которой определяются содержащим ее методом, поле связано с содержащим его типом. Листинг 3.1 может ссылаться на поле `_count` по его имени без дополнительного определения, поскольку поля находятся в области видимости в пределах своего определяющего класса. Но как насчет времени жизни? Мы знаем, что каждый вызов метода получает свой собственный набор локальных переменных. Сколько существует наборов полей класса? Есть несколько вариантов в зависимости от того, как вы определяете поле, но в нашем случае это один на экземпляр. Листинг 3.2 использует класс `Counter` из листинга 3.1, чтобы проиллюстрировать это. Я написал этот код в отдельном классе, чтобы продемонстрировать, что мы можем вызвать открытый метод класса `Counter` из других классов.

Листинг 3.2. Использование пользовательского класса

```
class Program
{
    static void Main(string[] args)
    {
        var c1 = new Counter();
        var c2 = new Counter();
        Console.WriteLine("c1: " + c1.GetNextValue());
        Console.WriteLine("c1: " + c1.GetNextValue());
        Console.WriteLine("c1: " + c1.GetNextValue());

        Console.WriteLine("c2: " + c2.GetNextValue());
```

```
        Console.WriteLine("c1: " + c1.GetNextValue());
    }
}
```

Здесь используется оператор `new` для создания новых экземпляров моего класса. Поскольку я использую `new` дважды, то получаю два объекта `Counter`, каждый из которых имеет собственное поле `_count`. Таким образом, мы получаем два независимых счетчика, что и показывает вывод программы:

```
c1: 1
c1: 2
c1: 3
c2: 1
c1: 4
```

Как и следовало ожидать, начинается отсчет, после чего при переключении на второй счетчик запускается новая последовательность. Но когда мы возвращаемся к первому счетчику, он продолжает с того места, где остановился. Это доказывает, что каждый экземпляр имеет свой собственный `_count`. Но что делать, если мы этого не хотим? Иногда вам захочется работать с информацией, которая не относится к какому-либо одному объекту.

Статические члены

Ключевое слово `static` позволяет нам объявить, что член не связан ни с одним конкретным экземпляром класса. Листинг 3.3 показывает модифицированную версию класса `Counter` из листинга 3.1. Я добавил два новых статических члена для отслеживания и отчета по всем экземплярам.

Листинг 3.3. Класс со статическими членами

```
public class Counter
{
    private int _count;
    private static int _totalCount;

    public int GetNextValue()
    {
        _count += 1;
        _totalCount += 1;
        return _count;
    }

    public static int TotalCount => _totalCount;
}
```

`TotalCount` сообщает о количестве, но не выполняет никакой работы — он просто возвращает значение, которое класс обновляет, и, как я объясню в разделе «Свойства» на с. 213, это делает его идеальным кандидатом на роль свойства, а не метода. Статическое поле `_totalCount` отслеживает общее количество вызовов `GetNextValue`, в отличие от нестатического `_count`, который считает вызовы текущего экземпляра. Обратите внимание, что я могу использовать это статическое поле внутри `GetNextValue` точно так же, как я использую нестатический `_count`. Разница в поведении станет очевидна, когда я добавлю строку кода, показанную в листинге 3.4, в конец метода `Main` в листинге 3.2.

Листинг 3.4. Использование статического свойства

```
Console.WriteLine(Counter.TotalCount);
```

Эта строка отобразит 5, сумму двух счетчиков. Чтобы получить доступ к статическому члену, я просто пишу `ClassName.MemberName`. Фактически в листинге 3.4 используются два статических члена — помимо свойства `TotalCount` моего класса он использует статический метод `WriteLine` класса `Console`.

Поскольку я объявил `TotalCount` как статическое свойство, содержащийся в нем код имеет доступ только к другим статическим членам. Если он попытается использовать нестатическое поле `_count` или вызовет нестатический метод `GetNextValue`, компилятор начнет возмущаться. Замена `_total Count` на `_count` в свойстве `TotalCount` приведет к следующей ошибке:

```
error CS0120: An object reference is required for the non-static  
field, method, or property Counter._count'
```

Поскольку нестатические поля связаны с конкретным экземпляром класса, C# должен знать, какой экземпляр использовать. С нестатическим методом или свойством это будет тот экземпляр, для которого был вызван сам метод или свойство. Поэтому в листинге 3.2 я написал `c1.GetNextValue ()` или `c2.GetNextValue ()`, чтобы показать, какой из двух объектов использовать. C# передал ссылку, хранящуюся в `c1` или `c2` соответственно, как неявный скрытый первый аргумент. Вы можете получить эту ссылку из кода внутри класса, используя ключевое слово `this`. Листинг 3.5 показывает альтернативный способ того, как мы могли бы написать первую строку `GetNextValue` из листинга 3.3, т. е. с явным указанием на то, что `_count` является членом экземпляра, для которого был вызван метод `GetNextValue`.

Листинг 3.5. Ключевое слово `this`

```
this._count += 1;
```

Явный доступ к члену через `this` иногда необходим из-за конфликтов имен. Хотя все члены класса находятся в области видимости любого кода в том же классе, код в методе не разделяет с классом пространство объявления. Из главы 2 вы помните, что пространство объявления — это область кода, в которой одно имя не должно ссылаться на две разные сущности, и, поскольку методы не разделяют свои имена с содержащим классом, вы можете объявлять локальные переменные и параметры методов, которые имеют то же имя, что и у членов класса. Это может легко случиться, если вы не используете соглашение, такое как предварение имен полей подчеркиванием. В этом случае вы не получите ошибку — локальные переменные и параметры попросту перекрывают членов класса. Но вы все равно можете добраться до членов класса, используя доступ через `this`.

Статические методы не могут использовать ключевое слово `this`, потому что они не связаны с каким-либо конкретным экземпляром.

Статические классы

Некоторые классы содержат только статические члены. В пространстве имен `System.Threading` есть несколько примеров, среди которых различные классы для работы с многопоточностью. Например, класс `Interlocked` обеспечивает атомарные неблокируемые операции чтение-изменение-запись; класс `LazyInitializer` содержит вспомогательные методы для такой отложенной инициализации, которая позволяет избежать двойной инициализации в многопоточных средах. Эти классы работают только через статические методы. Нет смысла создавать экземпляры этих типов, потому что в их экземплярах нет полезной информации, которую стоило бы хранить.

Вы можете объявить, что ваш класс предназначен для подобного использования, поместив ключевое слово `static` перед ключевым словом `class`. Это компилирует класс таким образом, который предотвращает создание его экземпляров. Любой, кто попытается создать экземпляры такого класса, явно не понимает, что делает, поэтому ошибка компилятора будет полезным стимулом обратиться к документации.

Вы можете объявить, что хотите иметь возможность вызывать статические методы определенных классов, не называя каждый раз сами эти классы.

Это может быть полезно, если вы пишете код, который активно использует статические методы, предоставляемые определенным типом. (Кстати, это не ограничивается статическими классами. Вы можете использовать эту технику с любым классом, который имеет статические члены, хотя он будет наиболее полезен в случае классов, все члены которых являются статическими.) В листинге 3.6 используются статический метод (`Sin`) и статическое свойство (`PI`) класса `Math` (в пространстве имен `System`). Он также использует статический метод `WriteLine` класса `Console`. (В этом и следующем примере я показываю весь исходный файл, потому что директивы `using` особенно важны.)

Листинг 3.6. Обычное использование статических членов

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Math.Sin(Math.PI / 4));
    }
}
```

Листинг 3.7 в точности такой же, но он не квалифицирует ни один из трех статических членов именем их определяющего класса.

Листинг 3.7. Использование статических членов без явной квалификации

```
using static System.Console;
using static System.Math;

class Program
{
    static void Main(string[] args)
    {
        WriteLine(Sin(PI / 4));
    }
}
```

Чтобы использовать эту более лаконичную альтернативу, вы должны с помощью `using static` объявить, какие классы вы хотите использовать подобным образом. Если использование директив `using` обычно определяет пространство имен, позволяя использовать типы в этом пространстве имен без квалификации, то использование директив `using static` указывает на класс, позволяя использовать его статические члены без квалификации.

Ссылочные типы

Любой тип, определенный с помощью ключевого слова `class`, будет ссылочным типом, что означает, что переменная этого типа не будет содержать данные, составляющие экземпляр типа; вместо этого она может содержать ссылку на экземпляр типа. Следовательно, присвоения не копируют объект, они просто копируют ссылку. Листинг 3.8 содержит почти тот же код, что и листинг 3.2, за исключением того, что вместо использования ключевого слова `new` для инициализации переменной `c2` он инициализирует ее копированием `c1`.

Листинг 3.8. Копирование ссылок

```
Counter c1 = new Counter();
var c2 = c1;
Console.WriteLine("c1: " + c1.GetNextValue());
Console.WriteLine("c1: " + c1.GetNextValue());
Console.WriteLine("c1: " + c1.GetNextValue());

Console.WriteLine("c2: " + c2.GetNextValue());

Console.WriteLine("c1: " + c1.GetNextValue());
```

Поскольку в этом примере `new` используется только один раз, существует только один экземпляр `Counter`, и обе переменные ссылаются на один и тот же экземпляр. Таким образом, мы получаем другой результат:

```
c1: 1
c1: 2
c1: 3
c2: 4
c1: 5
```

Это относится не только к локальным данным — если вы используете ссылочный тип для любого другого типа переменной, такой как поле или свойство, присваивание сработает таким же образом, копируя ссылку, а не весь объект. Это определяющая характеристика ссылочного типа, и по поведению он отличается от встроенных числовых типов из главы 2. В них каждая переменная содержит значение, а не ссылку на значение, поэтому присвоение обязательно ведет к копированию значения. (Копирование значений недоступно для большинства ссылочных типов — см. врезку «Копирование экземпляров».)

КОПИРОВАНИЕ ЭКЗЕМПЛЯРОВ

Некоторые языки семейства С определяют стандартный способ сделать копию объекта. Например, в C++ вы можете написать конструктор копирования и перегрузить оператор присваивания; в языке есть правила того, как это применяется при дублировании объекта. В C# некоторые типы можно копировать, и это не только встроенные числовые типы. Далее в этой главе вы узнаете, как определить тип `struct`, который является пользовательским значимым типом. `struct` всегда можно скопировать, и нет способа как-то это изменить: присвоение просто копирует все поля, а если какие-либо из них имеют ссылочный тип, то копирует ссылку. Это иногда называют «неполной» копией, потому что копируется только содержимое структуры, но не то, на что `struct` ссылается.

Нет встроенного механизма для создания копии экземпляра класса. Библиотека классов .NET определяет `ICloneable`, интерфейс для дублирования объектов, но он не обрел широкой поддержки. Проблема этого API в том, что он не определяет, как обрабатывать объекты со ссылками на другие объекты. Должен ли клон дублировать объекты, на которые он ссылается (полная копия), или просто скопировать ссылки (неполная копия)? На практике типы, которым требуется себя копировать, часто просто предоставляют для этого специальный метод, а не соответствуют какому-либо шаблону.

Мы можем написать код, который определяет, указывают ли две ссылки на одно и то же. В листинге 3.9 он показывает, как три переменные ссылаются на два счетчика с одинаковым `count`, а затем проверяет их тождественность. По умолчанию оператор `==` выполняет именно такую проверку на тождественность объектов, если его operandы являются ссылочными типами. Однако типам разрешено переопределять оператор `==`. Тип `string` изменяет поведение `==` для сравнения значений, поэтому, если вы передадите два различных строковых объекта в качестве operandов `==`, результат будет равен `true`, если они содержат идентичный текст. Если вы хотите принудительно проверить объекты на идентичность, вы можете использовать статический метод `object.ReferenceEquals`.

Листинг 3.9. Сравнение ссылок

```
var c1 = new Counter();
c1.GetNextValue();
Counter c2 = c1;
var c3 = new Counter();
c3.GetNextValue();
```

```
Console.WriteLine(c1.Count);
Console.WriteLine(c2.Count);
Console.WriteLine(c3.Count);
Console.WriteLine(c1 == c2);
Console.WriteLine(c1 == c3);
Console.WriteLine(c2 == c3);
Console.WriteLine(object.ReferenceEquals(c1, c1));
Console.WriteLine(object.ReferenceEquals(c1, c1));
Console.WriteLine(object.ReferenceEquals(c2, c2));
```

Первые три строки вывода подтверждают, что все три переменные относятся к счетчикам с одинаковым отсчетом:

```
1
1
1
True
False
False
True
False
False
```

Он также показывает, что, хотя все они имеют одинаковое значение счетчика, только `c1` и `c2` считаются одним и тем же объектом. Это потому, что мы присвоили `c1` в `c2`, что означает, что `c1` и `c2` будут ссылаться на один и тот же объект, следовательно, первое сравнение будет успешным. Но `c3` целиком и полностью относится к другому объекту (даже если имеет одинаковое значение), поэтому второе сравнение не достигает успеха. (Я использовал здесь сравнения с помощью `==` и `object.ReferenceEquals`, чтобы проиллюстрировать, что они в данном случае делают одно и то же, потому что `Counter` не переопределил оператор `==`.)

Попробуем проделать то же самое с `int` вместо `Counter`, как показано в листинге 3.10. (Он инициализирует переменные в несколько своеобразной манере, чтобы быть максимально похожим на код из листинга 3.9.)

Листинг 3.10. Сравнение значений

```
int c1 = new int();
c1++;
int c2 = c1;
int c3 = new int();
c3++;
```

```
Console.WriteLine(c1);
Console.WriteLine(c2);
Console.WriteLine(c3);
Console.WriteLine(c1 == c2);
Console.WriteLine(c1 == c3);
Console.WriteLine(c2 == c3);
Console.WriteLine(object.ReferenceEquals(c1, c1));
Console.WriteLine(object.ReferenceEquals(c1, c1));
Console.WriteLine(object.ReferenceEquals(c2, c2));
Console.WriteLine(object.ReferenceEquals(c1, c1));
```

Как и раньше, мы видим, что все три переменные имеют одинаковое значение:

```
1
1
1
True
True
True
False
False
False
False
```

Листинг также показывает, что тип `int` наделяет оператор `==` особым значением. В случае `int` этот оператор сравнивает значения, поэтому все три сравнения выполняются успешно. Но `object.ReferenceEquals` никогда не завершается успешно для значимых типов — я даже добавил дополнительное, четвертое сравнение, где безуспешно сравниваю `c1` с самим собой! Этот неожиданный результат продиктован тем, что нет никакого смысла выполнять сравнение ссылок с `int`, который не является ссылочным типом. Для последних четырех строк листинга 3.10 компилятор вынужден выполнить неявные преобразования из `int` в `object`: он обернул каждый аргумент `object.ReferenceEquals` в так называемую упаковку, которую мы рассмотрим в главе 7. Каждый аргумент получает отдельную упаковку, поэтому даже последнее сравнение не удается.

Есть еще одно различие между ссылочными типами и типами вроде `int`. По умолчанию любая переменная ссылочного типа может содержать специальное значение `null`, означающее, что переменная вообще не ссылается на какой-либо объект. Вы не можете присвоить это значение ни одному из встроенных числовых типов (см. врезку «`Nullable<T>`»).

NULLABLE<T>

.NET определяет тип-обертку `Nullable<T>`, который добавляет к значимым типам функционал допустимости содержания `NULL`. Хотя переменная `int` не может содержать `null`, `Nullable<int>` — может. Угловые скобки после имени типа указывают на то, что это обобщенный тип — вместо заполнителя `T` можно вписать различные типы, — и об этом я расскажу в главе 4.

Компилятор обрабатывает `Nullable<T>` особым образом. Он позволяет использовать более компактный синтаксис, так что можно записать просто `int?`. Когда внутри арифметических выражений появляются обнуляемые числа, компилятор обрабатывает их иначе, чем обычные значения. Например, если вы напишите `a + b`, где `a` и `b` являются `int?`, то результатом будет `int?`. Он будет равен `null`, если хотя бы один операнд равен `null`, а иначе будет содержать сумму значений. Это работает и если только один из операндов является `int?`, а другой обычным.

Несмотря на то что вы можете присвоить `null` переменной типа `int?`, это не ссылочный тип. Он больше похоже на сочетание `int` и `bool`. (Но как я опишу в главе 7, CLR иногда так хитро работает с `Nullable<T>`, что иногда это делает его больше похожим на ссылочный тип, чем на значимый тип.)

Если вы используете `null`-условные операторы, описанные в главе 2 (.? и ?[index]), для доступа к членам со значимым типом, полученное выражение будет иметь версию этого типа, допускающую значение `NULL`. Например, если `str` является переменной типа `string`, выражение `str?.Length` имеет тип `Nullable<int>` (или, если хотите, `int?`), потому что `Length` имеет тип `int`, но использование `null`-условного оператора означает, что выражение может получить значение `null`.

Избавляемся от `null` с помощью необнуляемых ссылок

Широким распространением нулевых ссылок в языках программирования мы обязаны ученому-компьютерщику Тони Хоару, который в 1965 году добавил их к очень влиятельному на тот момент языку ALGOL. Он уже извинился за это свое изобретение, которое назвал «ошибкой на миллиард долларов». Возможность того, что переменная ссылочного типа может содержать `null`, затрудняет определение того, насколько безопасно пытаться выполнить с ней какое-либо действие. (Программы C# выдают исключение `NullReferenceException`, если вы пытаетесь это проделать, что обычно приводит к сбою вашей программы. Исключения обсуждаются главе 8.) Некоторые современные языки программирования избегают использования обнуляемых ссылок по умолчанию, предлагая вместо этого какую-то систему для необязательных значений через явный механизм формального

согласия в системе типов. Как мы уже видели в случае с `Nullable<T>`, это уже имеет место для встроенных числовых типов (а также, как мы увидим, для любых пользовательских значимых типов, которые вы определяете), но до недавнего времени допустимость значения `NULL` не была опциональной для всех переменных ссылочного типа.

C# 8.0 вводит в язык важную новую функциональность, которая расширяет систему типов, позволяя различать ссылки, которые могут содержать `null`, и ссылки, которые не могут. Речь идет о ссылках, допускающих значение `null`, что звучит странно, потому что ссылки всегда могли содержать `null`, начиная еще с C# 1.0. Однако название происходит из того факта, что в разделях кода, которые используют эту функцию, допустимость содержания значения `NULL` становится опциональной: ссылка никогда не будет содержать `null`, если она явно не определена как допускающая значение `NULL`. По крайней мере, в теории.



Дать системе типов возможность различать допускающие и не допускающие `null` ссылки всегда было непростой задачей — это означает перекроить почти двадцать лет истории языка. Реальность такова, что C# не всегда может гарантировать, что не допускающая `null` ссылка никогда не будет содержать `null`. Тем не менее он способен гарантировать то, что соблюдены определенные ограничения и в целом значительно уменьшит шансы получить исключение `NullReferenceException` даже в случаях, когда это нельзя исключить полностью.

Сделать ссылки по умолчанию не допускающими значение `NULL` — это радикальный шаг, поэтому эта функция отключена до тех пор, пока вы явно ее не запросите. А поскольку ее включение может оказаться существенное влияние на готовый код, то этой функцией можно управлять очень тонко.

C# обеспечивает управление в двух измерениях, которые он называет контекстом с заметками о допустимости значения `NULL` и контекстом с предупреждениями о допустимости значения `NULL`. Каждая строка кода в программе на C# связана с одним контекстом каждого вида. По умолчанию весь ваш код находится в отключенном контексте с заметками о допустимости значения `NULL` и в отключенном контексте с предупреждениями о допустимости значения `NULL`. Вы можете изменить эти значения по умолчанию на уровне проекта. Вы также можете использовать директиву `#nullable` для изменения контекста с заметками о допустимости значения `NULL` на более

детальном уровне — по желанию сделать его разным для каждой строки, — и вы можете одинаково точно контролировать контекст с предупреждениями о допустимости значения `NULL` с помощью директивы `#pragma warning`. Так как же работают эти два контекста?

Контекст с заметками о допустимости значения `NULL` определяет, можем ли мы объявить определенное использование ссылочного типа допускающим значение `NULL` (например, поля, переменной или аргумента). Через отключенный контекст с заметками (по умолчанию) мы это сделать не можем, и все ссылки неявно допускают значение `null`. Официальная классификация описывает их как не учитывающие допустимость значения `null`, отличая их от ссылок, которые вы намеренно обозначили как допускающие `null`. Однако во включенном контексте с заметками мы уже можем выбирать. В листинге 3.11 показано, как именно это делается.

Листинг 3.11. Задание допустимости содержания `NULL`

```
string cannotBeNull = "Text";
string? mayBeNull = null;
```

Это должно выглядеть знакомо, потому что повторяет синтаксис функционала допустимости значения `NULL` для встроенных числовых типов и пользовательских типов-значений. Если вы просто напишете имя типа, это будет означать, что он не допускает `null`. Если вы хотите, чтобы он допускал `null`, вы добавляете `?`.

Здесь важно отметить, что во включенном контексте с заметками о допустимости значения `NULL` старый синтаксис получает новое поведение, а если вам нужно старое поведение, вам нужно использовать новый синтаксис. Это означает, что если вы возьмете существующий код, изначально написанный без какого-либо понятия о допустимости содержания `NULL`, и поместите его во включенный контекст с заметками, все переменные ссылочного типа будут по сути аннотированы как не допускающие `NULL` в противоположность тому, как компилятор обрабатывал точно такой же код раньше.

Самый прямой способ поместить код в контекст с заметками о допустимости значения `NULL` — это директива `#nullable enable`. Вы можете поместить ее в начало файла исходного кода, чтобы включить контекст для всего файла, или же можете использовать его более локально, сопроводив `#nullable restore` там, где нужно вернуть настройки по умолчанию для всего проекта. Само по себе это не приведет к видимым изменениям. Компилятор не будет

реагировать на эти аннотации, если контекст с предупреждениями о допустимости значения `NULL` отключен, а по умолчанию он отключен. Вы можете включить его локально с помощью `#pragma warning enable nullable` (а `#pragma warning restore nullable` восстанавливает значения по умолчанию для всего проекта). Вы можете управлять настройками по умолчанию в файле `.csproj`, добавив свойство `<Nullable>`. Листинг 3.12 устанавливает значения по умолчанию для включения контекста с предупреждениями о допустимости значения `NULL` и отключения контекста с заметками о допустимости значения `NULL`.

Листинг 3.12. Включение контекста с предупреждениями о допустимости значения `NULL` по умолчанию для всего проекта

```
<PropertyGroup>
    <Nullable>warnings</Nullable>
</PropertyGroup>
```

Это означает, что любые файлы, которые явно не включают контекст с заметками о допустимости значения `NULL`, будут работать в отключенном контексте. Контекст с заметками о допустимости значения `NULL`, но весь код будет в активированном контексте с предупреждениями о допустимости значения `NULL`, если он не отключен явно. Другие параметры для всего проекта: `disable` (по умолчанию), `enable` (использует контексты с предупреждениями и с заметками) и `annotations` (включить заметки, но не предупреждения).

Если вы включили контекст с заметками на уровне проекта, вы можете использовать `#nullable disable` для его отключения в отдельных файлах. Аналогично, если вы в любой форме включили контекст с предупреждениями на уровне проекта, вы можете отключить его с помощью `#pragma warning disable nullable`.

У нас есть все эти детальные элементы управления, чтобы упростить включение функционала проверки допустимости значения `NULL` для существующего кода. Если вы просто включите функцию для всего проекта разом, вы, скорее всего, столкнетесь с большим количеством предупреждений. На практике может иметь больше смысла поместить весь код в проекте во включенный контекст с предупреждениями, но не включать повсеместно аннотации, поскольку все ваши ссылки будут считаться опускающими проверку на возможность содержания `NULL`, пока вы не будете видеть никаких предупреждений. Затем можно начать перемещать код во включенный контекст с заметками по одному файлу за раз (или, если хотите, даже более мелкими фрагментами), внося необходимые изменения.

Конечная цель будет заключаться в том, чтобы довести весь код до такого состояния, чтобы можно было на уровне проекта полностью включить поддержку ссылок, не допускающих `null`.

Что компилятор делает для нас в коде, в котором мы полностью включили такую поддержку? Мы получаем две основные вещи. Во-первых, компилятор использует правила, аналогичные правилам определенного присваивания, чтобы гарантировать, что мы не пытаемся разыменовать метод, не проверив его на `null`. В листинге 3.13 показаны некоторые случаи, которые компилятор примет, и те, которые будут вызывать предупреждения во включенном контексте с предупреждениями, если предположить, что `mayBeNull` был объявлен во включенном контексте с заметками как допускающий `null`.

Листинг 3.13. Разыменование ссылки, допускающей `null`

```
if (mayBeNull != null)
{
    // Разрешено, потому что попасть сюда можно только в случае,
    // если mayBeNull не равно null
    Console.WriteLine(mayBeNull.Length);
}

// Разрешено, потому что значение проверяется на null и обрабатывается
Console.WriteLine (mayBeNull?.Length ?? 0);
// Компилятор предупредит об этом во включенном контексте
// с предупреждениями
Console.WriteLine(mayBeNull.Length);
```

Во-вторых, в дополнение к проверке того, является ли разыменование (использование `.` для доступа к члену) безопасным, компилятор предупредит вас, если вы попытаетесь присвоить ссылку, которая может быть нулевой, чему-то, что требует не допускающей `null` ссылки, или, если вы передаете ее в качестве аргумента методу, когда соответствующий параметр объявлен как не допускающий `null`.

Иногда вы будете сталкиваться с препятствиями в переводе всего кода в полностью включенные контексты допустимости значения `NULL`. Возможно, вы зависите от какого-то компонента, который вряд ли будет в обозримом будущем обновлен до поддержки контекстов допустимости `NULL`, или, возможно, существует сценарий, в котором консервативные правила безопасности C# ошибочно решают, что какой-то код небезопасен. Что можно сделать в этих случаях? Нежелательно и отключать предупреждения для всего проекта и оставлять код, набитый директивами `#pragma`. Но этому есть

альтернатива: можно сказать компилятору C#, что вы знаете что-то, чего он не знает. Если у вас есть ссылка на то, что по предположению компилятора может быть `null` (возможно, потому, что оно получено от компонента, который не поддерживает проверку на допустимость `NULL`), но у вас есть веские основания полагать, что он никогда не будет равен `null`, вы можете сообщить об этом компилятору с помощью оператора, допускающего `NULL`, который вы можете увидеть в конце второй строки листинга 3.14. Иногда его неофициально называют *оператором черт побери*, потому что восклицательный знак делает его несколько сердитым.

Листинг 3.14. Оператор, допускающий `NULL`

```
string? referenceFromLegacyComponent =
    legacy.GetReferenceWeKnowWontBeNull();
string nonNullableReferenceFromLegacyComponent =
    referenceFromLegacyComponent!;
```

Вы можете использовать оператор, допускающий `NULL`, в любом включенном контексте с заметками о допустимости значения `NULL`. По сути он преобразует допускающую `null` ссылку в не допускающую `null` ссылку. Затем уже можно перейти к разыменованию этой не допускающей `null` ссылки или использовать ее иным образом там, где допускающая `null` ссылка не будет разрешена без предупреждений компилятора.



Оператор, допускающий `NULL`, не проверяет свои входные данные. Если вы примените его в сценарии, где значение во время выполнения окажется равно `null`, он этого не обнаружит. Вместо этого вы получите ошибку во время выполнения в точке, где вы пытаетесь использовать ссылку.

В то время как оператор, допускающий `NULL`, может оказаться полезен на границе между кодом, учитывающим допустимость значения `NULL`, и старым кодом, который вы не контролируете, есть еще один способ сообщить компилятору, когда явно допускающее `null` выражение фактически не будет равно `null`, и это атрибуты, допускающие `NULL`. .NET определяет несколько атрибутов, которые можно использовать для аннотирования кода с целью указать, когда он будет возвращать `null`, а когда не будет. Рассмотрим код в листинге 3.15. Если вы не активируете функционал допускающего `null` ссылочного типа, он будет работать нормально, но если вы его включите, то получите предупреждение. (При этом используется словарь, тип коллекции, который подробно описан в главе 5.)

Листинг 3.15. Допустимость значения `NULL` и шаблон `Try` — перед допускающими `NULL` ссылочными типами

```
public static string Get(IDictionary<int, string> d)
{
    if (d.TryGetValue(42, out string s))
    {
        return s;
    }

    return "Not found";
}
```

При включенных предупреждениях о допустимости значения `NULL` компилятор будет жаловаться на `out string s`. Он справедливо скажет вам, что `TryGetValue` может пропустить `null` в аргумент `out`. (Этот тип аргумента обсуждается позже; он позволяет возвращать дополнительные значения помимо основного возвращаемого значения функции.) Эта функция проверяет, содержит ли словарь запись с указанным ключом. Если это так, он вернет `true` и поместит соответствующее значение в аргумент `out`, но если нет, он вернет `false` и установит для этого аргумента значение `null`. Мы можем изменить наш код, чтобы он отражал этот факт, поставив `?` после `out string`. Листинг 3.16 показывает эту модификацию.

Листинг 3.16. Использующий функционал допустимости значения `NULL` шаблон `Try`

```
public static string Get(IDictionary<int, string> d)
{
    if (d.TryGetValue(42, out string? s))
    {
        return s;
    }

    return "Not found";
}
```

Здесь можно ожидать новой проблемы. Наш метод `Get` возвращает `string`, а не `string?`, так как же `return s` может быть правильным? Мы ведь просто изменили наш код, чтобы указать, что `s` может быть `null`, поэтому не будет ли компилятор жаловаться, когда мы попытаемся вернуть это, возможно, равное `null` значение из метода, который объявляет, что он никогда не вернет `null`? Но на самом деле код компилируется. Компилятор принимает его, потому что знает, что `TryGetValue` установит аргумент `out` в `null`, только если он вернет `false`. Это означает, что компилятор знает, что, хотя тип переменной `s` — это

`string?`, он не будет равен `null` внутри тела оператора `if`. Он знает это благодаря атрибуту, допускающему `NULL`, примененному к определению метода `TryGetValue`. (Атрибуты описаны в главе 14.) Листинг 3.17 показывает атрибут в объявлении метода. (Этот метод является частью обобщенного типа, поэтому мы видим здесь `TKey` и `TValue`, а не типы `int` и `string`, которые я использовал в моих примерах. Глава 4 подробно обсуждает эту разновидность методов. В приведенных примерах `TKey` и `TValue`, по сути, являются `int` и `string`.)

Листинг 3.17. Атрибут, допускающий `NULL`

```
public bool TryGetValue(TKey key, [MaybeNullWhen(false)] out TValue value)
```

Эта аннотация позволяет C# понять, что значение может быть равно `null`, если `TryGetValue` возвращает `false`. Без этого атрибута листинг 3.15 скомпилировался бы успешно даже с включенными предупреждениями о допустимости значения `NULL`, потому что, написав `IDictionary <int, string>` (а не `IDictionary <int, string?>`), я указал, что мой словарь не допускает значения `null`. Обычно C# предполагает, что когда метод возвращает значение из словаря, он также создает экземпляр `string`. Но `TryGetValue` иногда ничего не возвращает, поэтому ему и нужна эта аннотация. Таблица 3.1 описывает различные атрибуты, которые вы можете применять, чтобы дать компилятору C# больше информации о том, что может, а что не может быть равным `null`.

Эти атрибуты были применены к большинству широко используемых частей библиотек классов .NET в .NET Core 3.0, чтобы уменьшить трения, связанные с принятием допускающих `NULL` ссылок.

Таблица 3.1. Атрибуты, допускающие `NULL`

Тип	Использование
AllowNull	Коду разрешено передавать <code>null</code> , даже если тип не допускает значение <code>NULL</code>
DisallowNull	Код не должен передавать <code>null</code> , даже если тип допускает значение <code>NULL</code>
MayBeNull	Нужно быть готовым, что это может вернуть <code>null</code> , даже если тип не допускает значение <code>NULL</code>
MaybeNullWhen	Используется только с параметрами <code>out</code> или <code>ref</code> ; вывод может быть равен <code>null</code> , если метод возвращает заданное значение типа <code>bool</code>
NotNullWhen	Используется только с параметрами <code>out</code> или <code>ref</code> ; вывод может не быть равен <code>null</code> , если метод возвращает заданное значение типа <code>bool</code>
NotNullIfNotNull	Если вы передадите не равное <code>null</code> значение в качестве аргумента для параметра, который этот атрибут называет, значение, возвращаемое целью этого атрибута, не будет равно <code>null</code>

Перемещение кода во включенные контексты с предупреждениями и заметками о допустимости значения `NULL` способно значительно повысить качество кода. Многие разработчики, которые переносят существующие базы исходного кода, в процессе часто обнаруживают ранее не найденные ошибки, и все благодаря дополнительным проверкам, которые выполняет компилятор. Тем не менее этот функционал не идеален. Есть две дыры, о которых стоит знать и которые связаны с тем, что функционал допустимости `NULL` изначально отсутствовал в системе типов. Во-первых, устаревший код порождает белые пятна — даже если весь ваш код находится во включенном контексте с заметками о допустимости значения `NULL`, но использует API, которые в нем не находятся, ссылки, которые он от них получает, будут игнорировать допустимость значения `NULL`. Если для спокойствия компилятора вам пришлось использовать оператор, допускающий `NULL`, всегда есть вероятность, что вы ошибаетесь и в результате вы получите `null` в переменной, не допускающей значения `NULL`. Вторая дыра еще неприятнее, поскольку вы можете столкнуться с ней и в самом новом коде, даже если вы изначально полностью включили этот функционал: в ряде хранилищ .NET при инициализации память заполняется нулевыми значениями. Если эти хранилища содержат ссылочные типы, их исходным значением будет `null`, и в настоящее время нет никакого способа, которым компилятор C# мог бы навязать им тип, не допускающий значения `NULL`. Эта проблема возникает с массивами. Посмотрите на листинг 3.18.

Листинг 3.18. Массивы и допустимость содержания `NULL`

```
var nullableStrings = new string?[10];
var nonNullableStrings = new string[10];
```

Этот код объявляет два массива строк. Первый использует `string?`, поэтому он допускает содержащие `NULL` ссылки. Второй — нет. Однако в .NET вам необходимо создать массивы, прежде чем вы сможете что-то в них поместить, а память вновь созданного массива всегда заполняется нулями. Это означает, что наш массив `nonNullableStrings` начнет существование с того, что заполнится значениями `null`. Из-за того, как работают массивы в .NET, избежать этого невозможно. Один из способов решить эту проблему — избегать прямого использования массивов. Если вы вместо массива используете `List<string>` (см. главу 5), он будет содержать только те элементы, которые вы добавили. В отличие от массива, `List<T>` нельзя при инициализации заполнить пустыми элементами. Но подобное решение возможно далеко не

всегда. Иногда вам просто нужно позаботиться о том, чтобы инициализировать все элементы в массиве.

Схожая проблема существует с полями в значимых типах, которые описаны в следующем разделе, когда они содержат поля ссылочного типа, т. е. ситуации, в которых невозможно предотвратить их инициализацию в `null`. Таким образом, функционал ссылок, допускающих значение `NULL`, не идеален. Тем не менее пользу от него сложно переоценить. Команды, которые внесли необходимые изменения в существующие проекты, сообщили, что сам этот процесс позволяет обнаруживать многие ранее скрытые ошибки. Это важный инструмент для улучшения качества вашего кода.

Хотя ссылки, не допускающие значения `NULL`, уменьшают одно из различий между ссылочными типами истроенными числовыми типами, другие важные различия все же остаются. Переменная типа `int` не является ссылкой на `int`. Он содержит значение `int` без какого-либо косвенного обращения. В некоторых языках этот выбор между поведением ссылочных типов и типов значений определяется тем, как вы используете тип, но в C# это фиксированная особенность типа. Каждый конкретный тип является либо ссылочным, либо значимым типом. Все встроенные числовые типы являются значимыми типами, как и `bool`, тогда как класс — всегда ссылочным. Но это отличие отсутствует в случае встроенных и пользовательских типов. Вы можете создавать собственные значимые типы.

Структуры

Иногда для пользовательского типа оказывается целесообразным такое же поведение, как у встроенных значимых типов. Наиболее очевидным примером может послужить пользовательский числовой тип. Хотя CLR предлагает различные внутренние числовые типы, некоторые виды вычислений требуют немного большей сложности, чем они могут дать. Например, многие научные и инженерные расчеты подразумевают работу с комплексными числами. Среда выполнения не содержит для них внутреннего представления, но библиотека классов поддерживает с помощью типа `Complex`. Если бы подобные числовые типы вели себя иначе, чем встроенные, от них было бы мало пользы. К счастью, это не так, потому что это — значимый тип. Чтобы задать собственный значимый тип, используйте ключевое слово `struct` вместо `class`.

Структура может содержать большинство тех же функций, что и класс: методы, поля, свойства, конструкторы и любые другие типы членов, поддерживаемые классами. И мы можем использовать те же ключевые слова для определения доступа к ним, такие как `public` и `internal`. Некоторые ограничения присутствуют, но в случае с простым типом `Counter`, который я написал ранее, я мог бы просто заменить ключевое слово `class` на `struct`. Однако это было бы не самым полезным преобразованием. Вспомните, что одно из основных различий между ссылочными типами (классами) и значимыми типами заключается в том, что у первых есть отличительные черты: я могу по желанию создать несколько объектов `Counter` для подсчета различных вещей. Но для значимых типов (встроенных или пользовательских) предполагается, что их можно свободно копировать. Если у меня есть экземпляр типа `int` (например, 4) и я храню его в нескольких полях, то не стоит ожидать, что конкретное значение существует само по себе: один экземпляр числа 4 неотличим от другого. Переменные, которые содержат значения, имеют свои собственные идентификаторы и время жизни, а вот значения, которые они содержат, — нет. Это отличается от того, как работают ссылочные типы: не только переменные, которые ссылаются на них, имеют идентификаторы и время жизни, но и объекты, на которые они ссылаются, имеют свои собственные идентификаторы и время жизни, которые не зависят от какой-либо конкретной переменной.

Если я добавлю единицу к значению `int`, равному 4, результатом будет совершенно другое значение `int`. Если я вызываю `GetNextValue()` экземпляра `Counter`, его счетчик увеличивается на единицу, но при этом он остается тем же экземпляром `Counter`. Поэтому, хотя замена `class` на `struct` в листинге 3.3 будет компилироваться, в действительности мы не хотим, чтобы наш тип `Counter` стал структурой. Листинг 3.19 показывает вариант получше.

Листинг 3.19. Простая структура

```
public struct Point
{
    private double _x;
    private double _y;
    public Point(double x, double y)
    {
        _x = x;
        _y = y;
    }
    public double X => _x;
    public double Y => _y;
}
```

GETHASHCODE

Метод `GetHashCode` содержат все типы .NET. Он возвращает `int`, который в некотором смысле представляет значение вашего объекта. Некоторые структуры данных и алгоритмы предназначены для работы с подобной упрощенной усеченной версией значения объекта. Например, хеш-таблица способна очень эффективно искать конкретную запись в очень большой таблице, если тип искомого значения содержит хорошую реализацию хеш-кода. На этом основаны некоторые из классов коллекций, описанных в главе 5. Детали реализации подобных алгоритмов выходят за рамки этой книги, но если в интернете вы введете в поиск «хеш-таблица», вы найдете достаточно много информации.

Правильная реализация `GetHashCode` должна соответствовать двум требованиям. Во-первых, независимо от того, какое число экземпляр возвращает в качестве своего хеш-кода, он должен продолжать возвращать тот же код, пока его собственное значение не изменится. Второе требование состоит в том, что два экземпляра, которые имеют равные значения, возвращаемые их методами `Equals`, должны возвращать один и тот же хеш-код. Любой тип, который не соответствует любому из этих требований, может породить код, использование в котором `GetHashCode` приведет к некорректной работе. Реализация метода `GetHashCode` по умолчанию для ссылочных типов соответствует первому требованию, но даже не пытается выполнить второе — выберите любые два объекта, которые используют реализацию по умолчанию, и большую часть времени они будут иметь разные хеш-коды. Это нормально, потому что реализация `Equals` по умолчанию для ссылочного типа всегда возвращает `true`, если вы сравниваете объект с самим собой, но именно поэтому вам необходимо переопределить `GetHashCode`, если вы переопределяете `Equals`. Значимые типы по умолчанию получают реализации `GetHashCode` и `Equals`, которые удовлетворяют обоим требованиям. Тем не менее они используют достаточно медленный механизм отражения (см. главу 13), поэтому обычно полезно написать свои собственные.

В идеале объекты, которые имеют разные значения, должны иметь и разные хеш-коды, но это не всегда возможно — `GetHashCode` возвращает `int`, который имеет конечное число возможных значений. (4 294 967 296, если быть точным.) Если ваш тип данных предлагает больше отдельных значений, то очевидно, что для каждого возможного значения будет невозможно создать отдельный хеш-код. Например, 64-битный целочисленный тип `long` очевидно поддерживает больше отдельных значений, чем `int`. Если вы вызываете `GetHashCode` для `long` со значением 0, в .NET 4.0 он возвращает 0 и вы получите тот же хеш-код для `long` со значением 4 294 967 297. Подобное дублирование называется *коллизией хеш-функции*, что является неизбежной реальностью нашей жизни. Код, который зависит от хеш-кодов, должен уметь решать эту проблему.

Правила не требуют, чтобы отображение значений в хеш-коды навсегда оставалось фиксированным — они должны быть согласованными только на протяжении

жизненного цикла процесса. Но вообще есть веские причины для их изменчивости. Злоумышленники, которые атакуют компьютерные онлайн-системы, иногда пытаются вызвать коллизии хеш-функции. Коллизии снижают эффективность алгоритмов на основе хеш-функции, поэтому атака, которая пытается перегрузить центральный процессор сервера, будет более эффективной, если сможет вызвать коллизии для значений, которые будут использоваться сервером в поиске на основе хеш-функции. Чтобы избежать этой проблемы, некоторые типы в библиотеке классов .NET преднамеренно меняют способ создания хеш-кодов при каждом перезапуске программы.

Поскольку коллизии хеш-функций неизбежны, их нельзя запретить правилами, что означает, что вы можете каждый раз возвращать одно и то же значение (например, 0) из `GetHashCode` независимо от фактического значения экземпляра. Хотя технически это не противоречит правилам, результатом будет отвратительная производительность при работе с хеш-таблицами и подобными вещами. В идеале вы будете стремиться минимизировать количество коллизий хеш-функций. Тем не менее, если вы не ожидаете, что в дальнейшем что-то будет критически зависеть от хеш-кода вашего типа, нет смысла тратить время на тщательную разработку хеш-функции, которая выдает хорошо распределенные значения. Иногда оправдан ленивый подход, при котором можно полагаться на единственное поле. Или можно положиться на кортеж, как это делает листинг 3.20, потому что кортежи умеют неплохо создавать хеш-коды для всех своих свойств. Или, если для вас допустимо ориентироваться на .NET Core 3.0, или .NET Standard 2.1, или более позднюю версию, вы можете использовать метод `HashCode.Combine`.

Этой структурой представлена точка в двумерном пространстве. И хотя, безусловно, можно представить, что нам требуются индивидуальные отдельные точки (и в этом случае нам бы понадобился `class`), вполне разумно хотеть иметь тип, подобный по поведению типу значения, содержащий местоположение точки.

Хотя в листинге 3.19 все в порядке и без этого, значения обычно поддерживают сравнение. Как упоминалось ранее, C# определяет значение по умолчанию оператора `==` для ссылочных типов: оно эквивалентно `object.ReferenceEquals`, который проверяет тождественность. Это не имеет смысла для значимых типов, поэтому C# по умолчанию не поддерживает оператор `==` для структуры. Вы не обязаны предоставлять для него определение, но все встроенные значимые типы это делают, поэтому, если мы пытаемся создать тип с аналогичными характеристиками, нам тоже следует это сделать. Если вы добавите оператор `==` самостоятельно, компилятор сообщит вам, что вы должны определить и соответствующий оператор `!=`. Вы можете подумать,

что C# определит `!=` как обратный `==`, так как они означают противоположное. Однако для определенных пар операндов некоторые типы будут возвращать `false` для обоих операторов, поэтому C# требует, чтобы мы независимо определили оба. Как показано в листинге 3.20, для определения пользовательского значения оператора мы используем ключевое слово `operator`, за которым следует собственно оператор, который мы хотим настроить. Листинг ниже определяет поведение для `==` и `!=`, которые нетрудно задать для нашего простого типа.

Листинг 3.20. Поддержка пользовательского сравнения

```
public static bool operator ==(Point p1, Point p2)
{
    return p1.X == p2.X && p1.Y == p2.Y;
}

public static bool operator !=(Point p1, Point p2)
{
    return p1.X != p2.X || p1.Y != p2.Y;
}

public override bool Equals(object obj)
{
    return obj is Point p2 && this.X == p2.X && this.Y == p2.Y;
}

public override int GetHashCode()
{
    return (X, Y).GetHashCode();
}
```

Если вы просто добавите операторы `==` и `!=`, вы обнаружите, что компилятор выдает предупреждения, рекомендующие вам определить два метода, называемые `Equals` и `GetHashCode`. `Equals`, — это стандартный метод, доступный для всех типов .NET, и, если вы определили пользовательское значение для `==`, следует убедиться, что `Equals` делает то же самое. Это показано в листинге 3.20, и, как вы можете видеть, он содержит ту же логику, что и оператор `==`, но выполняет некоторую дополнительную работу. Метод `Equals` допускает сравнение с любым типом, поэтому сначала мы проверяем, сравнивается ли наша точка с другой точкой. Я использовал шаблон типа для выполнения этой проверки, а также для помещения входящего аргумента `obj` в переменную типа `Point` в случае совпадения шаблона. Наконец, листинг 3.20

реализует `GetHashCode`, что нам тоже необходимо сделать, если мы реализуем `Equals`. Подробности см. во врезке «`GetHashCode`».

Код из листинга 3.20, добавленный к структуре из листинга 3.19, позволит нам провести несколько тестов. Листинг 3.21 работает аналогично листингам 3.9 и 3.10.

Листинг 3.21. Сравнение экземпляров структуры

```
var p1 = new Point(40, 2);
Point p2 = p1;
var p3 = new Point(40, 2);

Console.WriteLine($"{p1.X}, {p1.Y}");
Console.WriteLine($"{p2.X}, {p2.Y}");
Console.WriteLine($"{p3.X}, {p3.Y}");
Console.WriteLine(p1 == p2);
Console.WriteLine(p1 == p3);
Console.WriteLine(p2 == p3);
Console.WriteLine(object.ReferenceEquals(p1, p1));
Console.WriteLine(object.ReferenceEquals(p1, p1));
Console.WriteLine(object.ReferenceEquals(p2, p2));
Console.WriteLine(object.ReferenceEquals(p1, p1));
```

Запуск кода производит следующий вывод:

```
40, 2
40, 2
40, 2
True
True
True
False
False
False
False
```

Все три экземпляра имеют одинаковое значение. В случае `p2` это потому, что я инициализировал его, присвоив ему `p1`, а `p3` я создал с нуля, но с теми же аргументами. Затем у нас следуют первые три сравнения, которые, как вы помните, используют `==`. Поскольку листинг 3.20 определяет пользовательскую реализацию, которая сравнивает значения, все сравнения завершаются успешно. А все проверки `object.ReferenceEquals` безуспешны, потому что это значимый тип, как и `int`. Фактически это то же самое поведение, которое мы видели в листинге 3.10, где вместо

`Counter` использовался `int`. (Опять же, компилятор произвел неявную упаковку-преобразование, о чём мы поговорим в главе 7.) Таким образом, мы достигли нашей цели — определили тип с поведением, аналогичным встроенным значимым типам, таким как `int`.

Когда же писать значимые типы

Я уже показал ряд различий в наблюдаемом поведении `class` и `struct`, но, хотя я привел доводы, почему `Counter` — это плохой кандидат для `struct`, я не до конца объяснил, что было бы хорошим преобразованием. Короткий ответ: есть только два обстоятельства, в которых вам требуется значимый тип. Во-первых, если вам нужно представить что-то вроде значения, например число, структура, скорее всего, будет идеальным вариантом. Во-вторых, если вы определили, что структура имеет лучшие показатели производительности для сценария, в котором вы будете ее использовать, она может оказаться не идеальным, но все же хорошим выбором. Но стоит подробнее остановиться на плюсах и минусах. И я также коснусь удивительно устойчивого мифа о значимых типах.

При использовании ссылочных типов объект отличается от переменной, которая на него ссылается. Это может быть очень полезно, так как мы часто используем объекты в качестве моделей для реальных вещей со своими отличительными характеристиками. Но не без последствий для производительности. Время жизни объекта не обязательно напрямую связано со временем жизни переменной, которая на него ссылается. Вы можете создать новый объект, сохранить ссылку на него в локальной переменной, а затем скопировать эту ссылку в статическое поле. Метод, который первоначально создал объект, может затем завершиться, поэтому локальная переменной, которая изначально ссылалась на объект, больше не существует. Тем не менее объект должен продолжать существовать, потому что до него все еще можно добраться.

CLR делает все возможное, чтобы гарантировать, что память, которую занимает объект, не освобождается преждевременно. Но в конечном итоге, когда объект больше не используется, она все же освобождается. Это довольно сложный процесс (подробно описанный в главе 7), и приложения .NET могут в конечном итоге заставить CLR тратить значительное количество процессорного времени на отслеживание и ожидание, когда те или иные объекты выйдут из использования. Создание большого количества

объектов увеличивает эти расходы. Увеличение сложности также может определенным образом увеличить затраты на отслеживание объекта — если конкретный объект все еще существует лишь потому, что он достоин по очень запутанному пути, CLR может проходить по этому пути каждый раз при попытке определить, какая память еще используется. Каждый новый уровень косвенной адресации порождает необходимость дополнительной работы. Ссылка по определению является косвенной, поэтому каждая переменная ссылочного типа тратит время CLR.

Значимые же типы часто обрабатываются гораздо проще. Например, рассмотрим массивы. Если вы объявили массив какого-либо ссылочного типа, вы получите массив ссылок. Это очень гибкий подход, так как элементы могут быть равны null, а если вы хотите, то можете иметь несколько разных элементов, ссылающихся на один и тот же экземпляр. Но если вам в действительности нужен последовательный набор элементов, то эта гибкость чревата бессмысленным расходом времени. Для набора из 1000 экземпляров ссылочного типа требуется 1001 блок памяти: один блок содержит сам массив ссылок, а после него следуют 1000 объектов, на которые эти ссылки ссылаются. Но в случае значимых типов один блок может содержать все значения. Это упрощает управление памятью — массив либо все еще используется, либо нет, так что CLR не нужно проверять 1000 отдельных элементов.

Такая продуктивность может быть на руку не только в случае массивов. Определенные преимущества есть и в случае полей. Рассмотрим класс, который содержит 10 полей типа int. 40 байт, которые необходимы для хранения значений этих полей, могут находиться непосредственно в памяти, выделенной для экземпляра содержащего класса. Сравните это с 10 полями какого-либо ссылочного типа. Хотя эти ссылки могут храниться в памяти экземпляра объекта, объекты, на которые они ссылаются, будут самостоятельными объектами. Так что если все поля не равны NULL и все ссылаются на разные объекты, у вас имеются 11 блоков памяти — один для экземпляра, содержащего все поля, а затем по одному для каждого объекта, на которые эти поля ссылаются. На рис. 3.1 показаны различия между ссылками и значениями как для массивов, так и для объектов (с небольшим количеством примеров, потому что этот принцип применим даже к горстке экземпляров).

Значимые типы также иногда могут упростить работу со временем жизни. Зачастую память, выделенная для локальных переменных, может быть освобождена, как только метод завершается (хотя, как мы увидим в главе 9,

анонимные функции говорят нам, что все не так просто). Это означает, что память для локальных переменных часто может находиться в стеке, работа с которым обходится дешевле в плане производительности, чем с кучей. В случае ссылочных типов память для переменной — это только часть истории, и с объектом, на который она ссылается, так просто не справиться, потому что после выхода из метода он может оказаться доступным каким-то другим путем.

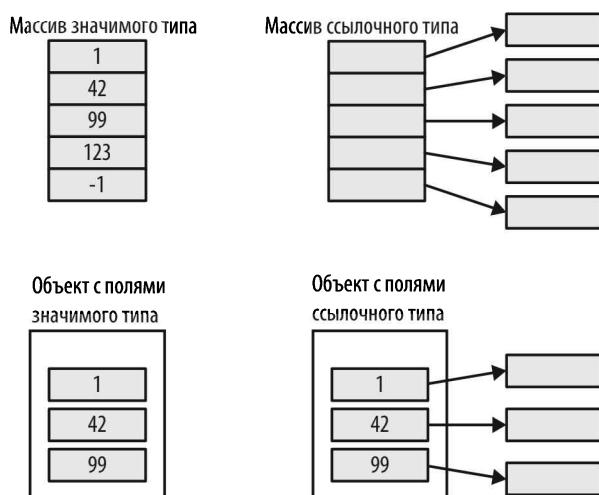


Рис. 3.1. Сравнение ссылок и значений

Фактически память, занятая значением, может быть восстановлена даже до возврата метода. Новые экземпляры значений часто перезаписывают старые. Например, C# обычно использует единственный фрагмент памяти для представления переменной независимо от того, сколько разных значений вы туда помещаете. Создание нового экземпляра значимого типа не обязательно означает выделение дополнительного объема памяти, тогда как для ссылочных типов новый экземпляр означает новый блок памяти в куче. Вот почему вполне нормально для каждой операции, которую мы выполняем со значимым типом, — например, целочисленного сложения или вычитания — создавать новый экземпляр.

Один из самых постоянных мифов о значимых типах гласит, что, в отличие от объектов, значения располагаются в стеке. Это правда, что объекты всегда существуют в куче, но значимые типы далеко не всегда обитают в стеке,

и даже в ситуациях, когда это так, это деталь реализации, а не фундаментальная особенность C#¹. На рис. 3.1 показаны два контрпримера. Значение `int` внутри массива типа `int[]` не находится в стеке; он расположено внутри блока, выделенного для массива в куче. Аналогично если класс объявляет нестатическое поле `int`, то значение этого `int` находится внутри блока, который выделен в куче для содержащего его экземпляра объекта. И даже локальные переменные значимого типа не обязательно попадают в стек. Например, оптимизация может позволить значению локальной переменной всегда оставаться в регистрах ЦП, без попадания в стек. И как вы увидите в главах 9 и 17, локальные переменные иногда могут пребывать в куче.

У вас может возникнуть соблазн обобщить предыдущие несколько абзацев словами «есть кое-какие сложные моменты, но, по сути, использование значимых типов более продуктивно». Но это было бы ошибкой. В некоторых ситуациях значимые типы обходятся значительно дороже. Помните, что главной особенностью значимого типа является то, что значения копируются при присваивании. Если значимый тип большой, это будет относительно накладно. Например, библиотека классов .NET определяет тип `Guid` для представления 16-байтовых глобальных уникальных идентификаторов, которые встречаются во многих местах Windows. Это структура, поэтому любой оператор присваивания, в котором используется `Guid`, будет запрашивать копию 16-байтовой структуры данных. Это, вероятно, окажется более затратной операцией, чем создание копии ссылки, поскольку для ссылок CLR использует реализацию, основанную на указателях; указатель обычно занимает 4 или 8 байт, но, что более важно, он вмещается в один регистр процессора.

К копированию значений ведет не только присваивание. Передача аргумента значимого типа методу может тоже потребовать копирования. Как это часто бывает, при вызове метода можно передать ссылку на значение, но, как мы увидим позже, это слегка ограниченный вид ссылки. Накладываемые им ограничения не всегда желательны, так что вы можете в конечном итоге решить, что затраты на копию предпочтительнее. Вот почему рекомендации Microsoft по проектированию предписывают не делать тип `struct`, если только он «не имеет размера экземпляра менее 16 байт» (правило, которое тип `Guid` технически нарушает, его размер составляет ровно 16 байт). Но это не жесткое правило, и на самом деле оно зависит от того, как вы будете его использовать, и поскольку более поздние версии C# обеспечивают большую

¹ У этого есть определенные исключения, описанные в главе 18.

гибкость для непрямого использования значимых типов, все чаще критичный к производительности код игнорирует это ограничение, а вместо этого заботится о минимизации копирования.

Значимые типы не будут автоматически эффективнее, чем ссылочные типы, и поэтому в большинстве случаев ваш выбор должен зависеть от требуемого поведения. Самый главный вопрос заключается в следующем: имеет ли для вас значение уникальность экземпляра? Другими словами, важно ли различие между одним объектом и другим? Для нашего примера с `Counter` ответ — да: если мы хотим, чтобы что-то считалось, проще всего, если счетчик — это что-то отдельное и уникальное. (В противном случае наш тип `Counter` ничем не отличается от `int`.) Но для нашего типа `Point` ответ уже отрицательный, так что он является вполне разумным кандидатом на роль значимого типа.

Важный и связанный с этим вопрос: содержит ли экземпляр вашего типа состояние, которое меняется со временем? Изменяемые значимые типы, как правило, представляют собой проблему, потому что слишком легко в конечном итоге работать с копией значения, а не с экземпляром, как вы изначально намеревались. (Важный пример этой проблемы я покажу позже, в разделе «Свойства и изменяемые значимые типы» на с. 218, и другой, когда я опишу `List<T>` в главе 5.) Так что, как правило, хорошая идея состоит в том, чтобы пользоваться неизменяемыми типами значений. Это не означает, что переменные этих типов не могут быть изменены; это просто означает, что для изменения переменной вы должны полностью заменить ее содержимое другим значением. Для чего-то простого, например типа `int`, отличие едва заметно, но оно важно для структур, содержащих несколько полей, таких как тип .NET `Complex`, который представляет числа, объединяющие вещественный и мнимый компоненты. Вы не можете изменить реальное или мнимое свойство существующего экземпляра `Complex`, так как тип является неизменным. И тип `Point`, показанный ранее, работает точно так же. Если полученное вами значение не соответствует ожидаемому, неизменность лишь означает, что вам нужно создать новое значение, поскольку вы не можете подкрутить существующий экземпляр.

Неизменность не обязательно означает, что вы должны создавать структуру — встроенный тип `string` является неизменным, а это класс. Тем не менее, поскольку C# зачастую не требуется выделять новую память для хранения новых экземпляров значимых типов, в их случае неизменность работает более эффективно, чем в случае классов в сценариях, где создается

много новых значений (например, в цикле)². Неизменность не является абсолютным требованием для структур — в библиотеке классов .NET есть некоторые многострадальные исключения. Но значимые типы обычно должны быть неизменяемыми, поэтому требование изменчивости обычно выступает хорошим признаком того, что вам нужен класс, а не структура.

Тип должен быть структурой только в том случае, если он по своей природе напоминает что-то, что является значимым типом. (В большинстве случаев он также должен быть довольно небольшим, поскольку передача больших типов по значению обходится дорого.) Например, в библиотеке классов .NET `Complex` представляет собой структуру, что неудивительно, поскольку это числовой тип, а все встроенные числовые типы являются значимыми типами. `TimeSpan` также является значимым типом, что имеет смысл, поскольку фактически это просто число, представляющее собой отрезок времени. В составе платформы пользовательского интерфейса WPF типы, используемые для простых геометрических данных, такие как `Point` и `Rect`, являются структурами. Но если вас грызут сомнения, пишите класс.

Гарантия неизменности

С версии C # 7.2 стало возможным объявить о своем намерении создать структуру только для чтения, добавив перед `struct` ключевое слово `readonly`, как показано в листинге 3.22. Она похожа на тип `Point`, показанный в листинге 3.19, но я сделал и несколько других изменений. Я не только добавил квалификатор `readonly`, но для уменьшения беспорядка также использовал автоматические свойства только для чтения. Я также добавил функцию-член по причинам, которые скоро станут понятны. Чтобы доступный только для чтения тип был полезен, он должен иметь конструктор, специальный член, который инициализирует поля и свойства. Более подробно я опишу их позже.

Применение ключевого слова `readonly` к структуре имеет два эффекта. Во-первых, компилятор C# будет следить за вашей честностью, предотвращая изменения снаружи или изнутри. Если вы объявили какие-либо поля, компилятор выдаст ошибку, если они не помечены `readonly`. Точно так же, если

² Здесь тип значения не поможет, потому что строки могут оказаться большими, а их передача по значению будет затратной. В любом случае это не может быть структура, потому что строки различаются по длине. Но это не то, что следует учитывать, потому что в C# вы не можете писать собственные типы данных переменной длины. Только строки и массивы имеют переменный размер.

вы попытаетесь определить настраиваемое свойство `auto` (описанное далее в этой главе), компилятор выдаст ошибку.

Листинг 3.22. Структура только для чтения

```
public readonly struct Point
{
    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }
    public double DistanceFromOrigin()
    {
        return Math.Sqrt(X * X + Y * Y);
    }
}
```

Во-вторых, структуры только для чтения содержат определенную оптимизацию. Если в каком-либо другом типе вы объявляете поле `readonly` (напрямую или косвенно с автоматическим свойством только для чтения), тип которого `readonly struct`, компилятор может избежать копирования данных, когда что-то использует это поле. Рассмотрим класс в листинге 3.23.

Листинг 3.23. Структура только для чтения в свойстве только для чтения

```
public class LocationRecord
{
    public LocationRecord(string label, Point location)
    {
        Label = label;
        Location = location;
    }

    public string Label { get; }
    public Point Location { get; }
}
```

Предположим, у вас есть переменная `r`, содержащая ссылку на `LocationRecord`. Что произойдет, если вы напишете выражение `r.Location.DistanceFromOrigin()`? Если рассуждать логически, мы просим `r.Location` получить

`Point`, а поскольку `Point` — это значимый тип, это повлечет за собой создание копии значения. Обычно C# генерирует код, который на самом деле делает копию, потому что в целом он не может знать, изменится ли `struct` после вызова какого-либо ее члена. Это называется защитными копиями, и они гарантируют, что подобные выражения не повлекут за собой неприятных сюрпризов, таких как изменение значения свойства или поля, которое должно быть только для чтения. Однако, поскольку `Point` является `readonly struct`, компилятору известно, что ему в данном случае не нужно создавать защитную копию. В этом случае компилятор C# или JIT-компилятор (или генератор кода АоТ) может безопасно оптимизировать этот код, вызывая `DistanceFromOrigin` непосредственно для значения, хранящегося в `LocationRecord`, без предварительного создания копии.



Если хотите, вы можете использовать `readonly struct` в доступных для записи полях и свойствах. Ключевое слово `readonly` гарантирует только то, что любое конкретное значение этого типа не изменится. Если вы хотите перезаписать существующее значение совершенно другим значением, это ваше дело.

Члены

Независимо от того, пишете ли вы класс или структуру, есть несколько различных типов элементов, которые можно поместить в пользовательский тип. Мы уже видели примеры некоторых из них, но давайте взглянем на них поближе.

За одним исключением (статические конструкторы), видимость можно указать для всех членов класса и структуры. Как тип может быть `public` или `internal`, так и каждый его член. Члены также могут быть объявлены `private`, что делает их видимыми только для кода внутри типа, и это видимость по умолчанию. И, как мы увидим в главе 6, наследование добавляет еще три уровня видимости для членов: `protected`, `protected internal` и `protected private`.

Поля

Вы уже видели, что поля являются именованными ячейками памяти, которые в зависимости от типа содержат либо значения, либо ссылки. По умолча-

нию каждый экземпляр типа получает собственный набор полей, но если вы хотите, чтобы поле было в единственном экземпляре, а не по одному на экземпляр, можно использовать ключевое слово `static`. Вы также можете применить ключевое слово `readonly` к полю, и это говорит о том, что оно может быть установлено только во время создания и не может быть изменено после.



Ключевое слово `readonly` не дает никаких гарантий. Существуют механизмы, с помощью которых можно добиться изменения значения поля `readonly`. Механизмы отражения, обсуждаемые в главе 13, предоставляют собой один способ, а небезопасный код, который позволяет напрямую работать с обычными указателями, — другой. Компилятор предотвратит случайное изменение поля, но при достаточной решимости вы можете обойти эту защиту. Но даже безо всяких уловок поле `readonly` может быть изменено в процессе работы конструктора.

C# предлагает ключевое слово с похожим на первый взгляд поведением: вы можете определить поле как `const`. Однако этот вариант предназначен для нескольких иных целей. Поле `readonly` инициализируется, а затем никогда не изменяется, тогда как поле `const` определяет значение, которое неизменно является одним и тем же. Поле `readonly` гораздо более гибкое: оно может быть любым типом, а его значение может быть вычислено во время выполнения, что означает, что вы можете пометить поля экземпляра класса или же поля `static` как `readonly`. Значение поля `const` определяется во время компиляции, что означает, что оно определено на уровне класса (так как у отдельных экземпляров нет возможности иметь разные значения). Это также ограничивает диапазон доступных типов. Для большинства ссылочных типов единственным поддерживаемым значением `const` является `null`, поэтому на практике обычно имеет смысл использовать `const` только с типами, которые внутренне поддерживаются компилятором. (В частности, если вы хотите использовать значения, отличные от `null`, тип `const` должен быть одним из встроенных числовых типов, `bool`, `string` или типом перечисления, как описано далее в этой главе.)

Это делает `const` более ограниченным, чем `readonly`, поэтому вы можете вполне ожидаемо спросить: а в чем его смысл? Что ж, хотя поле `const` и не такое гибкое, оно однозначно заявляет о неизменной природе значения. Например, класс .NET Math определяет константное поле типа `double`, называемое PI, которое содержит такое близкое приближение к матема-

тической константе PI, которое только может дать `double`. Это значение фиксировано навсегда, поэтому оно является константой в самом строгом смысле этого слова.

Когда дело доходит до менее постоянных значений, вы должны быть немного осторожнее с полями `const`; спецификация C# позволяет компилятору предполагать, что значение действительно никогда не изменится. Код, который читает значение поля `readonly`, извлекает значение из содержащей поле памяти прямо во время выполнения. Но когда вы используете поле `const`, компилятор может прочитать значение во время компиляции и скопировать его в IL, как если бы оно было литералом. Поэтому, если вы напишите библиотечный компонент, который объявляет поле `const`, и позже измените его значение, это изменение не обязательно будет воспринято кодом, использующим вашу библиотеку, если этот код не будет перекомпилирован.

Одним из преимуществ поля `const` является то, что оно подходит для использования в определенных контекстах, в которых поле `readonly` недоступно. Например, если вы хотите использовать шаблон константы (о шаблонах говорится в главе 2), скажем в метке `case` оператора `switch`, указанное вами значение должно быть зафиксировано во время компиляции. Таким образом, шаблон константы не может ссылаться на поле `readonly`, но вы можете использовать подходящее поле `const`. Вы также можете использовать поля `const` в выражении, определяющем значение другого поля `const` (при условии, что вы не вводите никаких циклических ссылок).

Поле `const` должно содержать выражение, определяющее его значение, как это показано в листинге 3.24.

Листинг 3.24. Поле `const`

```
const double kilometersPerMile = 1.609344;
```

Являясь обязательным для `const`, выражение в инициализаторе не обязательно для обычных и `readonly` полей³. Если вы опустите инициализирующее выражение, поле будет автоматически инициализировано значением по умолчанию. (Это `0` для числовых значений и его эквиваленты для других типов — `false`, `null` и т. д.) Структуры немного более ограничены, потому что, когда они инициализируются неявно, их поля экземпляра устанавливаются в `0`, поэтому вы не можете писать для них инициализаторы. Структуры

³ Если вы опускаете инициализатор для поля только для чтения, вы должны вместо этого установить его в конструкторе; в противном случае толку будет мало.

поддерживают инициализаторы для полей, не относящихся к экземпляру (т. е. полей `const` и `static`).

Если вы предоставляете выражение инициализатора для неконстантного поля, его не обязательно вычислять во время компиляции, поэтому оно может работать во время выполнения, выполняя вызов методов или чтение свойств. Конечно, подобный код может иметь побочные эффекты, поэтому важно знать порядок запуска инициализаторов.

Инициализаторы нестатических полей запускаются для каждого создаваемого вами экземпляра и выполняются в порядке их появления в файле непосредственно перед запуском конструктора. Инициализаторы статических полей выполняются не более одного раза независимо от того, сколько экземпляров типа вы создаете. Они также выполняются в том порядке, в котором объявлены, но точно определить, когда они будут выполняться, гораздо сложнее. Если в вашем классе нет статического конструктора, C# гарантирует запуск инициализаторов полей до первого обращения к полю в классе, но он не обязан ждать до последней минуты и сохраняет за собой право запускать инициализаторы полей уже в самом начале. (Точный момент, в который это происходит, варьировал в разных выпусках .NET.) Но если есть статический конструктор, то все немного яснее: инициализаторы статического поля запускаются непосредственно перед запуском статического конструктора, что порождает новые вопросы: что такое статический конструктор и когда он запускается? Поэтому нам лучше без отлагательств взглянуть на конструкторы.

Конструкторы

Вновь созданному объекту для работы может потребоваться какая-то информация. Например, класс `Uri` в пространстве имен `System` представляет собой унифицированный идентификатор ресурса (URI), такой как URL. Поскольку вся его цель состоит в том, чтобы содержать и предоставлять информацию о URI, не будет особого смысла иметь объект `Uri`, который не знает своего URI. Так что на самом деле невозможно создать его без указания URI. Если вы попробуете выполнить код листинга 3.25, то получите ошибку компилятора.

Листинг 3.25. Ошибка: не удалось передать Uri его URI

```
Uri oops = new Uri(); // Не будет компилироваться
```

Класс `Uri` определяет несколько конструкторов, членов, содержащих код, который инициализирует новый экземпляр типа. Если для работы определенного класса требуется определенная информация, вы можете отразить это требование с помощью конструкторов. Создание экземпляра класса почти всегда предполагает использование конструктора в какой-то момент, поэтому, если все ваши конструкторы запрашивают определенную информацию, разработчики вынуждены будут предоставить эту информацию, если они хотят использовать ваш класс⁴. Таким образом, все конструкторы класса `Uri` должны получить URI в той или иной форме.

Чтобы определить конструктор, вы сначала указываете видимость (`public`, `private`, `internal` и т. д.), а затем имя содержащего типа. Далее следует список параметров в скобках (который может быть пустым). В листинге 3.26 показан класс, который определяет один конструктор, требующий двух аргументов: один типа `decimal` и второй типа `string`. За списком аргументов следует блок, содержащий код. Таким образом, конструкторы очень похожи на методы, но вместо имени возвращаемого типа и имени метода они содержат имя содержащего типа.

Листинг 3.26. Класс с одним конструктором

```
public class Item
{
    public Item(decimal price, string name)
    {
        _price = price;
        _name = name;
    }
    private readonly decimal _price;
    private readonly string _name;
}
```

Этот конструктор довольно прост: он просто копирует свои аргументы в поля, и для многих конструкторов это максимум того, что они делают. Вы можете добавить туда столько кода, сколько захотите, но по обычай разработчики обычно ожидают, что конструктор будет делать очень мало и его основная задача — убедиться, что объект находится в допустимом начальном состоянии. Это может включать в себя проверку аргументов и вызов

⁴ Есть исключение. Если класс поддерживает функцию CLR, называемую сериализацией, объекты этого типа могут быть десериализованы непосредственно из потока данных, минуя конструкторы. Но даже здесь вы можете указать, какие данные требуются.

исключения, если возникла проблема, но не более того. Скорее всего, разработчики, использующие ваш класс, сильно удивятся, если вы напишите конструктор, который делает что-то особенное, например добавляет данные в базу данных или отправляет сообщение по сети.

Листинг 3.27 показывает, как использовать конструктор, принимающий аргументы. Мы попросту используем оператор `new`, передавая в качестве аргументов подходящие значения.

Листинг 3.27. Использование конструктора

```
var item1 = new Item(9.99M, "Hammer");
```

Вы можете определить несколько конструкторов, но должна быть возможность различать их: вы не можете определить два конструктора, оба из которых принимают одинаковое количество аргументов одинаковых типов, потому что у ключевого слова `new` не будет способа узнать, какой именно из них вы имели в виду.

Конструкторы по умолчанию и конструкторы с нулевым аргументом

Если вы вообще не определяете никаких конструкторов, C# предоставит конструктор по умолчанию, который эквивалентен пустому конструктору, который не принимает никаких аргументов. И если вы пишете структуру, он у вас будет, даже если вы определите другие конструкторы.



Хотя спецификация C# однозначно определяет конструктор по умолчанию как конструктор, сгенерированный для вас компилятором, имейте в виду, что есть еще одно широко используемое значение. Вы часто будете встречать термин *конструктор по умолчанию*, в применении для обозначения любого конструктора с общим доступом без параметров независимо от того, был ли он сгенерирован компилятором. В этом есть своя логика — с точки зрения кода, использующего класс, невозможно отличить конструктор, сгенерированный компилятором, от явного конструктора с нулевым аргументом. Поэтому, с этой точки зрения, если термин *конструктор по умолчанию* и означает что-то полезное, так это конструктор с общим доступом, который не принимает аргументов. Однако это не то, как спецификация C# определяет данный термин.

Сгенерированный компилятором конструктор по умолчанию не делает ничего, кроме нулевой инициализации полей, которая является отправной

точкой для всех новых объектов. Однако в некоторых ситуациях придется писать собственный конструктор без параметров. Вам может понадобиться, чтобы конструктор выполнил определенный код. Листинг 3.28 устанавливает поле `_id` на основе статического поля, которое увеличивается для каждого нового объекта с целью дать каждому экземпляру уникальный идентификатор. Это не требует передачи каких-либо аргументов, но включает в себя выполнение некоторого кода.

Листинг 3.28. Непустой конструктор с нулевым аргументом

```
public class ItemWithId
{
    private static int _lastId;
    private int _id;

    public ItemWithId()
    {
        _id = ++_lastId;
    }
}
```

Есть другой способ достижения того же эффекта. Я мог бы написать статический метод `GetNextId`, а затем использовать его в инициализаторе поля `_id`. Тогда мне вообще не нужно было бы писать этот конструктор. Тем не менее есть одно преимущество для помещения кода в конструктор: инициализаторам поля не разрешается вызывать собственные нестатические методы объекта, а конструкторам это позволительно. Все потому, что объект во время инициализации поля находится в незаконченном состоянии, соответственно, опасно вызывать его нестатические методы — они могут использовать поля, имеющие допустимые значения. Но объекту разрешено вызывать свои собственные нестатические методы внутри конструктора, потому что, хотя объект еще не полностью построен, он ближе к завершению, и поэтому уровень опасности ниже.

Есть и другие причины написания собственного конструктора с нулевым аргументом. Если вы определите хотя бы один конструктор для класса, это отключит генерацию конструктора по умолчанию. Если ваш класс уже предоставляет параметризованный конструктор, но вы все еще хотите иметь и конструктор без аргументов, вам нужно будет написать его, даже если он при этом окажется пустым. Возможен и вариант, когда вам требуется класс, единственным конструктором которого является пустой конструктор с нулевым аргументом, но с уровнем защиты, отличным от

`public`, который выдается по умолчанию. Например, если вы хотите сделать его `private`, чтобы только ваш код мог создавать экземпляры, то вам нужно задать конструктор явно, даже если он пуст, чтобы у вас было где указать уровень доступа.



Некоторые платформы могут использовать только классы, которые предоставляют конструктор с нулевым аргументом. Например, если вы создаете пользовательский интерфейс с помощью Windows Presentation Foundation (WPF), то классам, которые могут выступать в качестве пользовательских элементов интерфейса, обычно требуется именно такой конструктор.

В структурах конструкторы с нулевым аргументом работают немного иначе, потому что значимые типы должны поддерживать неявную инициализацию. Когда значимый тип используется в качестве поля какого-либо другого типа или типа элементов массива, память, в которой хранится значение, является частью содержащего объекта, так что когда вы создаете новый объект или массив, CLR всегда заполняет его память нулями. Это означает, что всегда можно инициализировать значение без передачи аргументов конструктору. Таким образом, в то время как C# удаляет конструктор по умолчанию для класса, когда вы добавляете конструктор, который принимает аргументы, он не делает этого для структур. Даже если он его скрыл, вы все равно можете косвенно вызывать эту неявную инициализацию, например путем создания одноэлементного массива этого типа:

```
MyStruct s = (new MyStruct[1])[0];
```

Поскольку неявная инициализация для структуры всегда доступна, компилятору нет никакого смысла скрывать соответствующий конструктор. C# не позволяет вам написать конструктор с нулевым аргументом для структуры, потому что существует очень много сценариев, в которых этот конструктор не будет работать. В большинстве случаев используется инициализация CLR нулями.

Цепочки конструкторов

Если вы пишите тип, который содержит несколько конструкторов, вы можете обнаружить, что они делают некоторое количество одинаковой работы, так как часто возникают задачи инициализации, которые должны выполнять все конструкторы. Класс в листинге 3.28 вычисляет числовой

идентификатор для каждого объекта в своем конструкторе, и если бы он содержал несколько конструкторов, им всем, возможно, потребовалось бы выполнять ту же работу. Перемещение этой части в инициализатор поля было бы одним из способов решения проблемы, но что, если это требуется только для части конструкторов? Вам может понадобиться, чтобы какая-то работа проделывалась почти всеми конструкторами, за исключением одного, который позволял бы указывать идентификатор явно, вместо того чтобы вычислять его. Подход с помощью инициализатора поля больше не будет работать должным образом, потому что вы хотите, чтобы отдельные конструкторы могли по выбору делать это или не делать. Листинг 3.29 показывает модифицированную версию кода из листинга 3.28, определяя два дополнительных конструктора.

Листинг 3.29. Необязательная цепочка конструкторов

```
public class ItemWithId
{
    private static int _lastId;
    private int _id;
    private string _name;

    public ItemWithId()
    {
        _id = ++_lastId;
    }

    public ItemWithId(string name)
        : this()
    {
        _name = name;
    }

    public ItemWithId(string name, int id)
    {
        _name = name;
        _id = id;
    }
}
```

Если вы взглянете на второй конструктор в листинге 3.29, за его списком параметров следует двоеточие, а затем вызов `this()`, который обращается к первому конструктору. Подобным образом конструктор может вызывать любой другой конструктор. Листинг 3.30 показывает другой способ организации всех трех конструкторов, иллюстрируя способ передачи аргументов.

Листинг 3.30. Цепочка конструкторов с использованием аргументов

```
public ItemWithId()
    : this(null)
{
}

public ItemWithId(string name)
    : this(name, ++_lastId)
{
}

public ItemWithId (string name, int id)
{
    _name = name;
    _id = id;
}
```

Конструктор с двумя аргументами теперь выступает своего рода главным конструктором — он единственный действительно выполняет какую-то работу. Другие конструкторы просто выбирают подходящие аргументы для этого основного конструктора. С некоторыми оговорками это более чистое решение, чем в предыдущих примерах, потому что работа по инициализации полей выполняется только в одном месте, а не в разных конструкторах, каждый из которых выполняет свою собственную часть.

Обратите внимание, что я назначил конструктору с двумя аргументами в листинге 3.30 видимость `private`. На первый взгляд может показаться немного странным определять способ создания экземпляра класса, а затем делать его недоступным, но при использовании цепочки конструкторов это имеет смысл. И есть другие сценарии, в которых конструктор с доступом `private` может оказаться кстати — мы можем захотеть написать метод, который делает клон существующего `ItemWithId`, и в этом случае такой конструктор будет действительно нужен. А оставляя его закрытым, мы сохраняем полный контроль над тем, как именно создаются новые объекты. Иногда бывает даже полезно сделать все конструкторы типа закрытыми, заставляя пользователей для получения экземпляра этого типа пользоваться тем, что иногда называют фабричным методом (статическим методом, который создает объект). Есть две общеизвестные причины для этого. Во-первых, если полная инициализация объекта требует такой дополнительной работы, которую неразумно производить в конструкторе (например, если нужно выполнить сравнительно медленную работу с использованием описанных

в главе 17 асинхронных функций языка, то вы не сможете поместить этот код в конструктор). Другой вариант — это когда вы хотите использовать наследование (см. главу 6) для предоставления нескольких вариантов типа, но при этом во время выполнения иметь возможность решать, какой именно тип будет возвращен.

Статические конструкторы

Конструкторы, которые мы рассмотрели до сих пор, запускаются при создании нового экземпляра объекта. Классы и структуры также могут определять и статический конструктор. Он выполняется не более одного раза за время существования приложения. Вы не вызываете его явно — C# гарантирует, что он запускается автоматически в какой-то момент перед первым использованием класса. Таким образом, в отличие от конструктора экземпляра, у вас нет возможности передавать ему аргументы. Поскольку статические конструкторы не могут принимать аргументы, в классе может быть только один такой конструктор. Кроме того, поскольку к ним никогда не обращаются явно, вы самостоятельно не назначаете какую-либо область видимости для статического конструктора. Листинг 3.31 показывает класс со статическим конструктором.

Листинг 3.31. Класс со статическим конструктором

```
public class Bar
{
    private static DateTime _firstUsed;
    static Bar()
    {
        Console.WriteLine("Bar's static constructor");
        _firstUsed = DateTime.Now;
    }
}
```

Так же как конструктор экземпляра переводит экземпляр в имеющее смысл начальное состояние, статический конструктор предоставляет возможность инициализировать любые статические поля.

Кстати, вы не обязаны гарантировать, что конструктор (статический или экземпляра) инициализирует каждое поле. Когда создается новый экземпляр класса, сначала все поля экземпляра устанавливаются в 0 (или эквивалентные значения, такие как `false` или `null`). Аналогично все статические поля типа обнуляются перед первым использованием класса. В отличие от

локальных переменных, вам нужно инициализировать поля, только если вы хотите установить для них значение, отличное от значения по умолчанию, т. е. нуля.

Даже тогда вам может не понадобиться конструктор. Инициализатора поля может оказаться достаточно. Однако не помешает точно знать, когда выполняются конструкторы и инициализация полей. Ранее я упоминал, что поведение варьирует в зависимости от наличия конструкторов, поэтому теперь, когда мы рассмотрели конструкторы более подробно, я наконец-то могу дать вам более полную картину инициализации. (Но это еще не все, потому что, как описывает глава 6, наследование добавляет еще одно измерение.)

Во время выполнения статические поля типа сначала будут установлены в 0 (или эквивалентные значения). Далее инициализаторы полей запускаются в том порядке, в котором они записаны в файле исходного кода. Этот порядок имеет значение, если инициализатор одного поля ссылается на другой. В листинге 3.32 поля `a` и `c` оба имеют одно и то же выражение для инициализации, но в итоге они имеют разные значения (1 и 42 соответственно) из-за порядка запуска инициализаторов.

Листинг 3.32. Упорядоченность статических полей

```
private static int a = b + 1;  
private static int b = 41;  
private static int c = b + 1;
```

Точный момент запуска инициализатора статического поля зависит от того, есть ли статический конструктор. Как уже упоминалось, в случае отсутствия такого конструктора время не определено — С# гарантирует их запуск не позднее первой попытки доступа к одному из полей типа, но оставляет за собой право запускать их сколь угодно рано. Наличие статического конструктора все меняет: в этом случае инициализаторы статических полей запускаются непосредственно перед конструктором. Так когда же запускается конструктор? Он будет вызван одним из двух событий в зависимости от того, что произойдет первым: создание экземпляра или доступ к любому статическому члену класса.

Для нестатических полей история аналогична: сначала все поля инициализируются равными 0 (или эквивалентными значениями), а затем инициализаторы полей запускаются в порядке их появления в исходном файле, и это

происходит еще до запуска конструктора. Конечно, различие заключается в том, что конструкторы экземпляров вызываются явно, поэтому ясно, когда происходит инициализация.

Для иллюстрации описанного поведения я написал класс `InitializationTestClass`, который можно увидеть в листинге 3.33. Класс имеет как статические, так и нестатические поля, и все они вызывают метод `GetValue` в своих инициализаторах. Этот метод всегда возвращает одно и то же значение, а именно 1, но при этом выводит сообщение, чтобы мы могли видеть, когда именно он вызывается. Класс также определяет конструктор экземпляра без аргументов и статический конструктор, которые, в свою очередь, выводят сообщения.

Листинг 3.33. Порядок инициализации

```
public class InitializationTestClass
{
    public InitializationTestClass()
    {
        Console.WriteLine("Constructor");
    }

    static InitializationTestClass()
    {
        Console.WriteLine("Static constructor");
    }

    public static int s1 = GetValue("Static field 1");
    public int ns1 = GetValue("Non-static field 1");
    public static int s2 = GetValue("Static field 2");
    public int ns2 = GetValue("Non-static field 2");

    private static int GetValue(string message)
    {
        Console.WriteLine(message);
        return 1;
    }

    public static void Foo()
    {
        Console.WriteLine("Static method");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main");
        InitializationTestClass.Foo();
        Console.WriteLine("Constructing 1");
        InitializationTestClass i = new InitializationTestClass();
        Console.WriteLine("Constructing 2");
        i = new InitializationTestClass();
    }
}
```

Метод `Main` выводит сообщение, вызывает статический метод, объявленный `InitializationTestClass`, а затем создает пару экземпляров. Запустив программу, я увижу следующий вывод:

```
Main
Static field 1
Static field 2
Static constructor
Static method
Constructing 1
Non-static field 1
Non-static field 2
Constructor
Constructing 2
Non-static field 1
Non-static field 2
Constructor
```

Обратите внимание, что инициализаторы статического поля и статический конструктор выполняются до того, как начнется вызов статического метода (`Foo`). Инициализаторы полей запускаются перед статическим конструктором и ожидаемо работают в порядке появления в исходном файле. Поскольку данный класс включает в себя статический конструктор, мы знаем, когда начнется статическая инициализация — она запускается при первом использовании этого типа, что в этом примере происходит, когда метод `Main` вызывает `InitializationTestClass.Foo`. Вы можете наблюдать, как это происходит непосредственно перед этой точкой, а не раньше, потому что наш метод `Main` выводит свое первое сообщение до того, как произойдет статическая инициализация. Если бы этот пример не имел статического конструктора, а имел только инициализаторы статического поля, не было бы никакой гарантии, что статическая инициализация произойдет в той

же самой точке; спецификация C# позволяет инициализации произойти раньше.

Следует быть осторожным с тем, что вы делаете в коде, который выполняется во время статической инициализации: он может выполниться раньше, чем вы ожидаете. Предположим, что ваша программа использует какой-либо механизм ведения журнала диагностики и вам необходимо настроить его при запуске программы, чтобы включить запись сообщений в нужное место. Всегда существует вероятность того, что код, выполняемый во время статической инициализации, может быть выполнен до того, как вам удастся это сделать. Это означает, что в этот момент ведение журнала диагностики еще не будет работать должным образом. Это может затруднить отладку кода. Даже если вы свяжете C# руки, предоставив статический конструктор, его тоже сравнительно легко вызвать раньше, чем вы предполагали. Использование любого статического члена класса вызовет его инициализацию, и вы можете оказаться в ситуации, когда ваш статический конструктор запускается инициализаторами статического поля в каком-то другом классе, у которого нет статического конструктора, а это может произойти еще до того, как запустится ваш метод `Main`.

Вы можете попытаться исправить это, инициализировав код журналирования в его собственной статической инициализации. Поскольку C# гарантирует запуск инициализации перед первым использованием типа, может показаться, что одно только это обеспечит завершение инициализации журналирования до статической инициализации любого кода, использующего систему ведения журнала. Однако и здесь есть потенциальная проблема: C# гарантирует только время начала статической инициализации для каждого конкретного класса. Но не гарантирует, что будет ждать ее окончания. Он и не может этого сделать, потому что если бы мог, то код, такой как, например, мы видим в листинге 3.34, поставил бы его в невозможное положение.

Листинг 3.34. Круговые статические зависимости

```
public class AfterYou
{
    static AfterYou()
    {
        Console.WriteLine("AfterYou static constructor starting");
        Console.WriteLine("AfterYou: NoAfterYou.Value = " +
NoAfterYou.Value);
        Value = 123;
```

```
        Console.WriteLine("AfterYou static constructor ending");
    }

public static int Value = 42;
}

public class NoAfterYou
{
    static NoAfterYou()
    {
        Console.WriteLine("NoAfterYou static constructor starting");
        Console.WriteLine("NoAfterYou: AfterYou.Value: = " +
                           AfterYou.Value);
        Value = 456;
        Console.WriteLine("NoAfterYou static constructor ending");
    }

    public static int Value = 42;
}
```

В этом примере наблюдается круговая связь двух типов: оба имеют статические конструкторы, которые пытаются использовать статическое поле, определенное другим классом. Поведение будет зависеть от того, какой из этих двух классов программа попытается использовать первым. В программе, которая сначала использует `AfterYou`, я наблюдаю следующий вывод:

```
AfterYou static constructor starting
NoAfterYou static constructor starting
NoAfterYou: AfterYou.Value: = 42
NoAfterYou static constructor ending
AfterYou: NoAfterYou.Value = 456
AfterYou static constructor ending
```

Как и следовало ожидать, статический конструктор `AfterYou` запускается первым, потому что это класс, который пытается использовать моя программа. Он выводит свое первое сообщение, но затем пытается использовать поле `NoAfterYou.Value`. Это означает, что теперь должна начаться статическая инициализация `NoAfterYou`, поэтому мы видим первое сообщение от его статического конструктора. Он пытается извлечь поле `AfterYou.Value`, несмотря на то что статический конструктор `AfterYou` еще не завершен. (Он получил значение, установленное инициализатором поля 42, а не значение 123, установленное статическим конструктором.) Это разрешено, потому что правила упорядочивания регулируют только начало статической инициа-

лизации и не гарантируют, когда она завершится. Если бы они отвечали за полную инициализацию, этот код не смог бы продолжить выполнение — статический конструктор `NoAfterYou` не смог бы двигаться вперед, потому что статическое конструирование `AfterYou` еще не завершено, но и оно не могло бы двигаться вперед, потому что ожидало бы окончания статической инициализации `NoAfterYou`.

Мораль этой истории в том, что вы не должны возлагать слишком большие надежды на статическую инициализацию. Может оказаться трудно предсказать точный порядок, в котором все будет происходить.



Пакет NuGet `Microsoft.Extensions.Hosting` обеспечивает гораздо лучший способ решения проблем инициализации с помощью класса `HostBuilder`. Это выходит за рамки этой главы, но его стоит отыскать и изучить.

Деконструкторы

В главе 2 мы узнали, как разобрать кортеж на составные части. Но деконструкция может пригодиться не только для кортежей. Вы можете включить деконструкцию для любого типа, который вы пишете, добавив нужный член `Deconstruct`, как показано в листинге 3.35.

Листинг 3.35. Включение деконструкции

```
public readonly struct Size
{
    public Size(double w, double h)
    {
        W = w;
        H = h;
    }

    public void Deconstruct(out double w, out double h)
    {
        w = W;
        h = H;
    }

    public double W { get; }
    public double H { get; }
}
```

C# распознает этот шаблон метода с именем `Deconstruct` и списком аргументов `out` (который будет подробно описан в следующем разделе), позволяя вам использовать тот же синтаксис деконструкции, что и в случае с кортежами. Листинг 3.36 использует этот метод для извлечения составляющих значений `Size`, чтобы как можно более кратко описать выполняемые вычисления.

Листинг 3.36. Использование пользовательского деконструктора

```
static double DiagonalLength(Size s)
{
    (double w, double h) = s;
    return Math.Sqrt(w * w + h * h);
}
```

Типы с деконструктором также могут использовать сопоставление с шаблоном позиции. В главе 2 показано, как для сопоставления кортежей можно использовать в шаблоне очень похожий на деконструкцию синтаксис. Любой тип с пользовательским деконструктором может использовать похожий синтаксис. В листинге 3.37 используется пользовательский деконструктор типа `Size` для определения различных шаблонов для `Size` в выражении `switch`.

Вспомните из главы 2, что шаблоны позиции являются рекурсивными: каждая позиция в шаблоне содержит вложенный шаблон. Так как `Size` деконструируется на два элемента, каждый шаблон позиции имеет две позиции для размещения дочерних шаблонов. Листинг 3.37 по-разному использует шаблоны константы, сброс и шаблоны типа.

Листинг 3.37. Шаблон позиции с использованием пользовательского деконструктора

```
static string DescribeSize(Size s) => s switch
{
    (0, 0) => "Empty",
    (0, _) => "Extremely narrow",
    (double w, 0) => $"Extremely short, and this wide: {w}",
    _ => "Normal"
};
```

Чтобы использовать деконструктор в шаблоне, C# должен во время компиляции знать тип, который будет деконструирован. Это можно увидеть в листинге 3.37, где входные данные для выражения `switch` имеют тип `Size`. Если вход шаблона позиции имеет тип `object`, компилятор предположит, что вы вместо этого пытаетесь сопоставить кортеж. Так будет происходить, если вы явно не назовете тип, как в листинге 3.38.

Листинг 3.38. Шаблон позиции с явным типом

```
static string Describe(object o) => o switch
{
    Size(0, 0) => "Empty",
    Size(0, _) => "Extremely narrow",
    Size(double w, 0) => $"Extremely short, and this wide: {w}",
    Size(_) => "Normal shape",
    _ => "Not a shape"
};
```

Хотя компилятор обрабатывает `Deconstruct` специальным образом, на что опираются данные примеры, с точки зрения среды выполнения это всего лишь обычный метод. Так что сейчас самое время более подробно рассмотреть методы.

Методы

Методы — это именованные части кода, которые могут возвращать результат и принимать аргументы. C# проводит довольно общее различие между *параметрами* и *аргументами*: метод определяет список ожидаемых входных данных — параметров — и код внутри метода ссылается на эти параметры по имени. Значения, видимые в коде, могут отличаться при каждом вызове метода, а термин *аргумент* относится к конкретному значению, назначенному параметру в конкретном вызове.

Как вы уже видели, когда присутствует спецификатор доступа, такой как `public` или `private`, то он появляется в начале объявления метода. Необязательное ключевое слово `static`, если оно вообще есть, располагается следующим. После этого в объявлении метода указывается тип возвращаемого значения. Как и во многих языках семейства C, вы можете написать методы, которые ничего не возвращают, помещая ключевое слово `void` вместо возвращаемого типа. Внутри метода вы используете ключевое слово `return`, за которым следует выражение, определяющее значение для возврата методом. В случае использования `void` вы можете для завершения метода использовать ключевое слово `return` без выражения, хотя это и не обязательно, потому что когда выполнение достигает конца такого метода, оно завершается автоматически. Обычно вы используете `return` в методе с `void` только тогда, когда ваш код решает, что должен завершиться досрочно.

Передача аргументов по ссылке

В C# методы могут непосредственно возвращать только один элемент. Если вы хотите вернуть несколько значений, вы, конечно, можете сделать этот

элемент кортежем. В качестве альтернативы можно обозначить, что параметры предназначены для вывода, а не для ввода. Листинг 3.39 возвращает два значения, оба получены целочисленным делением. Основным возвращаемым значением является частное, но также возвращается и остаток, что происходит с помощью последнего параметра, который был аннотирован ключевым словом `out`. Поскольку кортежи были введены только в C# 7, в то время как параметры с `out` присутствовали с самого начала, `out` используется гораздо чаще. Например, вы увидите множество методов, следующих схеме, аналогичной `int.TryParse`, где тип возвращаемого значения является логическим значением, указывающим на успех или неудачу, при этом фактический результат передается через параметр `out`.

Листинг 3.39. Возврат нескольких значений с помощью `out`

```
public static int Divide(int x, int y, out int remainder)
{
    remainder = x % y;
    return x / y;
}
```

В листинге 3.40 показан один из способов вызова метода с параметром `out`. Вместо предоставления выражения, как это делается с аргументами для нормальных параметров, мы написали ключевое слово `out`, за которым следует объявление переменной. Это создает новую переменную и инициализирует ее значением, которое метод возвращает через этот самый параметр `out`. Таким образом, в нашем случае получается новая переменная `r`, инициализированная в 1.

Листинг 3.40. Помещение результата параметра `out` в новую переменную

```
int q = Divide(10, 3, out int r);
```

Переменная, объявленная в аргументе `out`, следует обычным правилам области видимости, поэтому в листинге 3.40 `r` будет оставаться в области действия до тех пор, пока там остается `q`. Менее очевидно, что `r` доступна и в остальной части выражения. Листинг 3.41 использует это, чтобы попытаться интерпретировать текст как целое число, возвращая результат, если все прошло успешно, и аварийное значение 0, если синтаксический анализ не удался.

Листинг 3.41. Использование результата параметра `out` в том же выражении

```
int value = int.TryParse(text, out int x) ? x : 0;
```

Когда вы передаете аргумент `out`, передается ссылка на локальную переменную. Когда в листинге 3.40 вызывается `Divide` и когда этот метод присваивает значение переменной `remainder`, в действительности он присваивает его переменной `r` вызывающего метода. Это `int`, который является значимым типом, поэтому он обычно не передается по ссылке, так что такая ссылка ограничена в возможностях по сравнению с любым ссылочным типом. Например, вы не можете объявить поле в классе, которое может содержать ссылку такого рода, потому что локальная переменная `r` перестанет существовать, когда выйдет из области видимости, тогда как экземпляр класса может бесконечно долго оставаться в блоке кучи⁵. С# должен гарантировать, что вы не можете поместить ссылку на локальную переменную в то, что способно пережить переменную, на которую ссылается.



Методы, аннотированные ключевым словом `async` (описанным в главе 17), не могут иметь аргументов `out`. Это связано с тем, что асинхронные методы могут неявно возвращать управление вызывающей стороне до своего завершения, продолжая выполнение через некоторое время. Это, в свою очередь, означает, что вызывающий метод мог также завершиться до того, как `async` снова запустится, и в этом случае переменные, переданные по ссылке, могут уже не существовать на момент, когда асинхронный код будет готов их установить. То же ограничение применяется и к анонимным функциям (описанным в главе 9). Тем не менее оба вида методов могут передавать аргументы `out` в вызываемые ими методы.

Вы не обязаны всегда объявлять новую переменную для каждого аргумента `out`. Как показано в листинге 3.42, после `out` вы можете указать имя уже существующей переменной. (Когда-то это был единственный способ использовать аргументы `out`, поэтому обычно можно увидеть код, который объявляет новую переменную в отдельном операторе непосредственно перед его использованием в качестве аргумента `out`, даже если вариант, показанный в листинге 3.40, выглядит проще.)

Листинг 3.42. Помещение результата параметра `out` в существующую переменную

```
int r, q;  
q = Divide(10, 3, out r);  
Console.WriteLine($"3: {q}, {r}");  
q = Divide(10, 4, out r);  
Console.WriteLine($"4: {q}, {r}");
```

⁵ CLR называет такой тип ссылки управляемым указателем, чтобы отличать его от типа ссылки, которая ссылается на объект в куче. К сожалению, терминология С# менее точна: обе ссылки называются просто ссылками.



При вызове метода с параметром `out` мы должны явно указать, что знаем, как метод использует аргумент. Независимо от того, используем ли мы существующую переменную или объявляем новую, мы должны использовать ключевое слово `out` в месте вызова, как и в объявлении. (Некоторые языки семейства C не проводят никакого визуального различия между вызовами, которые передают значения, и вызовами, которые передают ссылки, но семантика сильно отличается, поэтому в C# это более явно.)

Иногда потребуется вызвать метод с аргументом `out`, который на самом деле не нужен. К примеру, вам нужно только основное возвращаемое значение. Как показано в листинге 3.43, после ключевого слова `out` можно поставить одно подчеркивание. Это скажет C#, что вам не требуется результат. (Это относительно новая функция, поэтому в старых кодовых базах довольно часто можно увидеть код, который вводит переменную, единственное предназначение которой состоит в том, чтобы разместить в ней ненужный результат.)

Листинг 3.43. Отказ от результата параметра `out`

```
int q = Divide(10, 3, out _);
```



Вам следует избегать использования `_` (одно подчеркивание) в качестве имени чего-либо в C#, потому что это способно помешать компилятору интерпретировать его как сброс. Если локальная переменная с этим именем находится в области видимости, запись `out` `_`, начиная с C# 1.0, указывает на то, что вы хотите присвоить результат `out` этой переменной, поэтому для обратной совместимости текущие версии C# должны сохранять такое поведение. Вы можете использовать эту форму сброса, только если в области действия нет символа с именем `_`.

Ссылка `out` требует передачи информации от метода обратно к вызывающей стороне, поэтому, если вы попытаетесь написать метод, который завершается без присвоения чего-либо всем своим аргументам `out`, вас ждет ошибка компилятора. Для проверки этого C# использует правила явного присваивания, упомянутые в главе 2. (Это требование не применяется, если метод выдает исключение вместо завершения.) Существует родственное ключевое слово `ref`, которое имеет аналогичную ссылочную семантику, но позволяет передавать информацию в двух направлениях. С аргументом `ref` создается впечатление, что метод имеет прямой доступ к переменной, которую передал вызывающий объект, — мы можем прочитать его текущее

значение, а также изменить его. (Вызывающая сторона обязана убедиться, что переменные, переданные через `ref`, содержат значение перед выполнением вызова, поэтому в данном случае метод не обязан устанавливать его перед возвратом.) Если вы вызываете метод с параметром, аннотированным `ref` вместо `out`, вы должны на стороне вызова четко указать, что хотите передать в качестве аргумента ссылку на переменную, как показано в листинге 3.44.

Листинг 3.44. Вызов метода с аргументом `ref`

```
long x = 41;  
Interlocked.Increment(ref x);
```

Есть третий способ использовать в аргументе косвенную адресацию: применить ключевое слово `in`. (Это было новшеством C# 7.2.) Если `out` позволяет информации исходить из метода, `in` позволяет ей только поступать туда. Это похоже на аргумент `ref`, но когда вызываемому методу не разрешено изменять переменную, на которую ссылается аргумент. Это может показаться излишним: если нет способа получить информацию обратно через аргумент, зачем передавать ее по ссылке? Аргумент `in int` не выглядит намного полезнее обычного аргумента `int`. На самом деле его и не следует использовать с `int`. Он нужен только при работе с относительно большими типами. Как вы знаете, значимые типы обычно передаются по значению, т. е. при передаче значения в качестве аргумента должна быть сделана копия. Ключевое слово `in` позволяет избежать копирования, передавая вместо этого ссылку. Раньше иногда использовали ключевое слово `ref`, чтобы избежать создания копий данных, но это чревато риском того, что метод может изменить значение вопреки желанию вызывающей стороны. С помощью `in` мы получаем ту же семантику «только вход», которую мы получаем при передаче значений обычным способом, но с потенциальным выигрышем в эффективности, так как не нужно передавать все значение.

Следует использовать `in` только для типов, которые больше, чем указатель. Вот почему аргумент `in int` бесполезен. `Int` имеет длину 32 бита, поэтому от передачи ссылки на `int` мы ничего не выигрываем. В 32-битном процессе этой ссылкой будет 32-битный указатель, так что в итоге мы получаем небольшую дополнительную бесполезную работу, связанную с косвенным использованием значения через ссылку. В 64-битном процессе такой ссылкой будет 64-битный указатель, поэтому нам пришлось бы передать в метод больше данных, чем если бы воспользовались `int` напрямую! (Иногда CLR может встроить метод и избежать затрат на создание указателя, но это означает, что в лучшем случае `in int` и `int` будут стоить одинаково. А по-

скольку `in` существует исключительно в целях производительности, он не очень полезен для небольших типов, таких как `int`.)

Листинг 3.45 определяет довольно большой значимый тип. Он содержит четыре значения `double`, каждое из которых имеет размер 8 байт, поэтому каждый экземпляр этого типа занимает 32 байта. Руководящие принципы проектирования .NET всегда рекомендовали избегать создания значимых типов такого большого размера, и главная причина этого заключается в том, что передача их в качестве аргументов затратна. Однако ключевое слово `in` способно снизить эти затраты, а это означает, что в некоторых случаях может иметь смысл определять такую большую структуру.

Листинг 3.45. Большой значимый тип

```
public readonly struct Rect
{
    public Rect(double x, double y, double width, double height)
    {
        X = x;
        Y = y;
        Width = width;
        Height = height;
    }

    public double X { get; }
    public double Y { get; }
    public double Width { get; }
    public double Height { get; }
}
```

В листинге 3.46 показан метод, который вычисляет площадь прямоугольника, представленного типом `Rect`, определенным в листинге 3.45. Мы действительно не хотим копировать все 32 байта ради вызова этого очень простого метода, тем более что он использует только половину данных из `Rect`. Поскольку этот метод аннотирует свой параметр с помощью `in`, такого копирования не произойдет: аргумент будет передан по ссылке, что на практике означает необходимость передачи только указателя, т. е. либо 4, либо 8 байт, в зависимости от того, выполняется код в 32-битном или 64-битном процессе.

Листинг 3.46. Метод с параметром `in`

```
public static double GetArea(in Rect r) => r.Width * r.Height;
```

Вы можете ожидать, что вызов метода с параметрами `in` потребует от стороны вызова указать, что она знает, что аргумент будет передан по ссылке через `in` перед аргументом, как нам нужно было написать `out` или `ref` на стороне вызова для двух других стилей передачи по ссылке. Но, как показано в листинге 3.47, это необязательно. Если вы хотите явно указать вызов по ссылке, вы можете это сделать, но, в отличие от `ref` и `out`, компилятор все равно просто передаст аргумент по ссылке, даже если вы не добавите `in`.

Листинг 3.47. Метод с параметром `in`

```
var r = new Rect(10, 20, 100, 100);
double area = GetArea(in r);
double area2 = GetArea(r);
```

Ключевое слово `in` является необязательным на стороне вызова, поскольку определение такого параметра, как `in`, является лишь оптимизацией производительности, которая не меняет поведение, в отличие от `out` и `ref`. Microsoft хотела, чтобы разработчики имели возможность внести совместимое с остальным исходным кодом изменение, в котором существующий метод изменяется путем добавления `in` к параметру. Это серьезное изменение на уровне двоичного представления, но в сценариях, при которых вы уверены, что людям в любом случае потребуется все перекомпилировать (например, когда весь код находится под вашим контролем), ради производительности может оказаться полезным внести такое изменение.

Конечно, как и во всех подобных улучшениях, следует измерить производительность до и после изменения, чтобы увидеть, достигнут ли ожидаемый эффект.

Хотя только показанные примеры работают как положено, они кроют в себе ловушку для неосторожных. Все работает только потому, что я пометил `struct` в листинге 3.45 как `readonly`. Если бы вместо определения моего собственного `Rect` я использовал очень похожую структуру с тем же именем из пространства имен `System.Windows` (часть инфраструктуры пользовательского интерфейса WPF), листинг 3.47 не спас бы меня от копирования. Он бы скомпилировался и произвел правильные результаты во время выполнения, но все это не дало бы никакого выигрыша в производительности. Все потому, что `System.Windows.Rect` не только для чтения. Ранее я обсуждал защитные копии, создаваемые C# при использовании поля `readonly`, содержащего изменяемый значимый тип. Здесь применяется тот же принцип, потому что входящий аргумент в действительности доступен только для чтения: код,

передающий аргументы, ожидает, что они не будут меняться, если только они явно не помечены как `out` или `ref`. Таким образом, компилятор должен убедиться, что аргументы `in` не меняются, даже если вызываемый метод обладает ссылкой на переменную вызывающего. Когда рассматриваемый тип уже только для чтения, компилятору не требуется выполнять никакой дополнительной работы. Но если это изменяемый значимый тип и если метод, которому был передан этот аргумент, в свою очередь, вызывает метод для этого значения, компилятор генерирует код, который делает копию, и вызывает метод для нее, потому что он никак не может узнать, способен ли метод изменить значение. Вам может показаться, что компилятору следует делать это принудительно, не позволяя методу с параметром `in` делать что-либо, что могло бы изменить значение. Однако на практике это означало бы, что он вообще не сможет вызывать какие-либо методы значения — в общем случае компилятор не способен определить, может ли конкретный вызов метода изменить значение. (И даже если сегодня он этого не делает, возможно, начнет в будущей версии библиотеки, которая определяет тип.) Поскольку свойства являются замаскированными методами, это сделало бы аргументы `in` ненужными. Это подводит нас к простому правилу:



Следует использовать `in` только со значимыми типами `readonly`, поскольку изменяемые значимые типы могут перечеркнуть все преимущества в плане производительности. (Изменяемые значимые типы — плохая идея в любом случае.)

C# 8.0 добавляет функционал, который способен немного ослабить это ограничение. Он позволяет применять ключевое слово `readonly` к членам, так что они могут объявлять, что не будут изменять значение, членом которого являются. Это позволяет избежать упомянутых защитных копий в случае изменяемых значений.

Ключевые слова `out` и `ref` можно использовать и со ссылочными типами. Это может показаться излишним, но иногда это полезно. Так можно реализовать двойную косвенную адресацию — метод получает ссылку на переменную, которая содержит ссылку. Когда вы передаете аргумент ссылочного типа методу, то метод получает доступ к любому объекту, который вы выбрали для передачи ему.

Хотя метод может использовать члены этого объекта, он обычно не в силах заменить его другим объектом. Но если вы пометите аргумент ссылочного

типа с помощью `ref`, у метода будет доступ к вашей переменной, поэтому он может заменить его ссылкой на совершенно другой объект.

Технически возможно, чтобы конструкторы также имели параметры `out` и `ref`, хотя это весьма необычно. Также следует внести ясность относительно того, что квалификаторы `out` или `ref` являются частью сигнатуры метода (или конструктора). Вызывающая сторона может передать аргумент `out` (или `ref`) тогда и только тогда, когда параметр был объявлен как `out` (или `ref`). Вызывающая сторона не может в одностороннем порядке принять решение передать аргумент посредством ссылки в метод, который его не ожидает.

Ссылочные переменные и возвращаемые значения

Теперь, когда вы увидели различные способы передачи методу ссылки на значение (или ссылки на ссылку), у вас может возникнуть вопрос, а можете ли вы получить эти ссылки другими способами. Можете, как показано в листинге 3.48, но с некоторыми ограничениями.

Листинг 3.48. Локальная переменная `ref`

```
string rose = null;
ref string rosaIndica = ref rose;
rosaIndica = "smell as sweet";
Console.WriteLine($"A rose by any other name would {rose}");
```

В этом примере объявляется переменная с именем `rose`, а после нее еще одна, с типом `ref string`. Ключевое слово `ref` здесь имеет точно такой же эффект, как и в параметре метода: указывает, что переменная является ссылкой на другую переменную. Поскольку код инициализирует ее с помощью `ref rose`, переменная `rosaIndica` является ссылкой на эту переменную `rose`. Поэтому когда код присваивает `rosaIndica` значение, оно попадает в переменную `rose`, на которую ссылается `rosaIndica`. Когда последняя строка прочитает значение переменной `rose`, в ней будет значение, записанное предыдущей строкой.

Итак, каковы же ограничения? Как вы видели ранее в случае `ref` и `out`, C# должен гарантировать, что вы не поместите ссылку на локальную переменную в то, что способно пережить переменную, на которую ссылается. Таким образом, вы не можете использовать это ключевое слово для поля. Статические поля существуют до тех пор, пока загружен их определяющий тип (обычно до завершения процесса), а поля — члены классов существуют в куче, что позволяет им пережить любой отдельно взятый вызов метода. (Это верно и для большинства структур. `ref struct` это не касается, но

даже в них в настоящее время не поддерживается ключевое слово `ref` для поля.) Даже в тех случаях, когда вам кажется, что время жизни — это не проблема (например, потому что цель ссылки сама по себе является полем в объекте), оказывается, что среда выполнения просто не поддерживает сохранение ссылок такого типа в поле или в качестве типа элемента массива. Менее очевидно то, что это также означает невозможность использования локальной переменной `ref` в контексте, где C# будет хранить переменную в классе. Это исключает их использование в асинхронных методах и итераторах, а также предотвращает их захват анонимными функциями (которые описаны в главах 17, 5 и 9 соответственно).

Хотя типы не могут определять поля с помощью `ref`, они могут определять методы, которые возвращают ссылку в стиле `ref` (и поскольку свойства являются скрытыми методами, получатель свойства также может возвращать ссылку). Как всегда, компилятор C# должен гарантировать, что ссылка не сможет пережить то, к чему она относится, поэтому будет препятствовать использованию этой функции в тех случаях, когда не может быть уверен, что сможет применить это правило. В листинге 3.49 показаны различные варианты использования возвращаемых типов `ref`, некоторые из которых компилятор принимает, а некоторые — нет.

Листинг 3.49. Допустимое и недопустимое использование возвращаемых значений `ref`

```
public class Referable
{
    private int i;
    private int[] items = new int[10];

    public ref int FieldRef => ref i;

    public ref int GetArrayElementRef(int index) => ref items[index];

    public ref int GetBackSameRef(ref int arg) => ref arg;

    public ref int WillNotCompile()
    {
        int v = 42;
        return ref v;
    }

    public ref int WillAlsoNotCompile()
    {
        int i = 42;
        return ref GetBackSameRef(ref i);
```

```
    }

    public ref int WillCompile(ref int i)
    {
        return ref GetBackSameRef(ref i);
    }
}
```

Методы, которые возвращают ссылку на значение `int`, являющееся полем или элементом массива, разрешены, потому что ссылки `ref` всегда могут ссылаться на элементы внутри объектов в куче. (Они просто не могут пребывать *внутри* них.) Объекты кучи могут существовать до тех пор, пока они необходимы (и сборщик мусора, рассматриваемый в главе 7, знает о таких ссылках и постарается, чтобы объекты кучи, на внутренности которых указывают ссылки, оставались живыми). И такой метод может вернуть любой из своих аргументов `ref`, потому что вызывающий объект уже был обязан убедиться, что они останутся действительными в течение всего времени вызова. Однако метод не может вернуть ссылку на одну из своих локальных переменных, потому что в тех случаях, когда эти переменные в конечном итоге оказываются в стеке, при завершении метода фрейм стека перестает существовать. Было бы проблемой, если бы метод был способен вернуть ссылку на переменную в уже не существующем фрейме стека.

Правила становятся немного более хитрыми, когда дело доходит до возврата ссылки, полученной из другого метода. Последние два метода в листинге 3.49 пытаются вернуть ссылку, возвращенную `GetBackSameRef`. Один работает, а другой — нет. Такой результат ожидаем: `WillAlsoNotCompile` следует отклонить по той же причине, по которой был отклонен `WillNotCompile`; оба пытаются вернуть ссылку на локальную переменную, но `WillAlsoNotCompile` просто пытается скрыть это, используя другой метод, `GetBackSameRef`. В подобных случаях компилятор C# делает консервативное предположение о том, что любой метод, который возвращает `ref` и принимает один или несколько аргументов `ref`, может вернуть ссылку в один из этих аргументов. Поэтому компилятор запрещает вызов `GetBackSameRef` в `WillAlsoNotCompile` на том основании, что тот может вернуть ссылку на ту же локальную переменную, которая была передана по ссылке. (И это в данном случае правильно. Но компилятор отклонил бы любой вызов такой формы, даже если рассматриваемый метод возвращал бы ссылку на что-то совершенно иное.) Но он позволяет `WillCompile` возвращать `ref`, возвращаемый `GetBackSameRef`, потому что в этом случае передаваемая ссылка является одной из тех, что нам напрямую разрешено возвращать.

Как и в случае с аргументами `in`, основная причина использования `ref` в качестве возвращаемых значений заключается в том, что они могут обеспечить большую производительность во время выполнения за счет избегания копирования. Вместо того чтобы возвращать значение целиком, методы такого типа могут просто возвращать указатель на существующее значение. Это также дает возможность вызывающей стороне изменять то, на что он ссылается. Скажем, в листинге 3.49 мне позволяет присвоить значение свойству `FieldRef`, даже если это свойство доступно только для чтения. Отсутствие задающего метода в данном случае не имеет значения, поскольку его тип — это `ref int`, который является допустимой целью присвоения. Итак, написав `r.FieldRef = 42;` (где `r` имеет тип `Referable`), я могу изменить поле `i`. Аналогично ссылка, возвращаемая `GetArrayElementRef`, может использоваться для изменения соответствующего элемента в массиве. Если вам это не нужно, можно сделать возвращаемый тип `ref readonly`, а не просто `ref`. В этом случае компилятор не позволит использовать итоговую ссылку в качестве цели присвоения.



Вы должны использовать возвращаемый `ref readonly` только со структурой `readonly`, потому что в противном случае вы столкнетесь с той же проблемой защитной копии, которую мы наблюдали ранее.

Необязательные аргументы

Вы можете сделать аргументы, отличные от `out` и `ref`, необязательными, задав значения по умолчанию. Метод в листинге 3.50 определяет значения, которые должны иметь аргументы, если вызывающая сторона их не предоставляет.

Листинг 3.50. Метод с необязательными аргументами

```
public static void Blame(string perpetrator = "the youth of today",
    string problem = "the downfall of society")
{
    Console.WriteLine($"I blame {perpetrator} for {problem}.");
}
```

Этот метод может быть вызван с одним аргументом, двумя или без аргументов вообще. Листинг 3.51 предоставляет только первый, беря проблемные аргументы из значений по умолчанию.

Листинг 3.51. Пропуск одного аргумента

```
Blame("mischievous gnomes");
```

Обычно при вызове метода аргументы указываются по порядку. Но что, если вы хотите вызвать метод в листинге 3.50, но хотите передать значение только для второго аргумента, используя значение по умолчанию для первого? Вы не можете просто оставить первый аргумент пустым — если вы попытаетесь написать `Blame(, "everything")`, то компилятор выдаст ошибку. Вместо этого можно указать имя аргумента, который хотите передать, с помощью синтаксиса, показанного в листинге 3.52. C# заполнит опущенные аргументы указанными значениями по умолчанию.

Листинг 3.52. Указание имени аргумента

```
Blame(problem: "everything");
```

Очевидно, что так аргументы можно опускать только тогда, когда вызываются методы, в которых определены значения аргументов по умолчанию. Однако вы можете указывать имена аргументов при вызове любого метода — иногда это может быть полезно, даже если вы не опускаете никаких аргументов. При чтении кода это само по себе упрощает понимание того, для чего нужны эти аргументы. Это особенно полезно, если вы столкнулись с API, который принимает аргументы типа `bool`, но сразу не понятно, что они означают. В листинге 3.53 создается `StreamReader` (описанный в главе 15), и этот конкретный конструктор принимает множество аргументов. Достаточно ясно, что представляют собой первые два, но оставшиеся три, скорее всего, будут загадкой для любого, кто читает код, если только он не запомнил все 11 перегрузок конструктора `StreamReader`. (Синтаксис объявления `using`, показанный здесь, описан в главе 7.)

Листинг 3.53. Загадочные аргументы

```
using var r = new StreamReader(stream, Encoding.UTF8, true, 8192, false);
```

Имена аргументов здесь не требуются, но если мы их включим, как в листинге 3.54, станет намного проще понять, что делает код.

Листинг 3.54. Улучшение ясности путем именования аргументов

```
using var r = new StreamReader(stream, Encoding.UTF8,
    detectEncodingFromByteOrderMarks: true, bufferSize: 8192,
    leaveOpen: false);
```

До C# 7.2, если вы начинали именовать аргументы, останавливаться было уже нельзя. Вам не позволили написать код, как в листинге 3.55. Основанием для этого ограничения было то, что, поскольку именованные аргументы по-

зволяют передавать аргументы в другом порядке, чем это было объявлено, компилятор не мог установить связь между позициями аргументов и позициями параметров. Однако это обоснование кажется подозрительным, потому что даже когда мы передаем все аргументы по порядку, мы все равно можем захотеть использовать имена аргументов исключительно для улучшения ясности. В любом случае теперь можно поступить так, как показано в листинге 3.55, и назвать только второй аргумент, сделав его назначение более очевидным, но не называть остальные аргументы, если и без этого достаточно ясно, что они делают.

Листинг 3.55. Выборочное именование аргументов

```
using var w = new StreamWriter(filepath, append: true, Encoding.UTF8);
```

Важно понимать, как C# реализует значения аргументов по умолчанию, потому что это влияет на написание библиотек. Когда вы вызываете метод без предоставления всех аргументов, как в листинге 3.52, компилятор генерирует код, который передает полный набор аргументов в обычном порядке. Он по сути переписывает ваш код, добавляя аргументы, которые вы пропустили. Значение этого состоит в том, что если вы напишите библиотеку, которая определяет значения аргументов по умолчанию, подобные этим, то у вас возникнут проблемы, когда вы захотите изменить значения по умолчанию. Код, скомпилированный со старой версией библиотеки, будет иметь на стороне вызова старые значения по умолчанию и не получит новые, пока не будет перекомпилирован.

Иногда вам будет встречаться альтернативный механизм, позволяющий опускать аргументы, а также избегать вставки значений по умолчанию на стороне вызова, и это перегрузка. Это слегка неестественный термин для довольно простой идеи о том, что одно имя или символ может иметь несколько значений. Фактически мы уже наблюдали эту технику на примере конструкторов — в листинге 3.30 я определил один главный конструктор, который выполнял реальную работу, а затем два других конструктора, которые его вызвали. Мы можем использовать тот же прием и с методами, как показано в листинге 3.56.

Листинг 3.56. Перегруженный метод

```
public static void Blame(string perpetrator, string problem)
{
    Console.WriteLine($"I blame {perpetrator} for {problem}.");
}
```

```
public static void Blame(string perpetrator)
{
    Blame(perpetrator, "the downfall of society");
}

public static void Blame()
{
    Blame("the youth of today", "the downfall of society");
}
```

В некотором смысле это немного менее гибкий механизм, чем значения аргументов по умолчанию, так как код, вызывающий метод `Blame`, больше не имеет способов указать значение для аргумента `problem` при выборе `perpetrator` по умолчанию (хотя было бы достаточно легко решить это с помощью добавления метода с другим именем). С другой стороны, перегрузка метода предлагает два потенциальных преимущества: она позволяет вам выбирать значения по умолчанию во время выполнения, если это необходимо, а также предоставляет способ сделать аргументы `out` и `ref` необязательными. Они требуют ссылки на локальные переменные, поэтому нет способа определить их значение по умолчанию, но всегда можно представить перегрузки с этими аргументами или без них, если это необходимо. И вы можете использовать смесь двух техник и полагаться в основном на необязательные аргументы, а перегрузки использовать только для того, чтобы иметь возможность опустить аргументы `out` или `ref`.

Подсчет переменных в параметрах с помощью ключевого слова `params`

Некоторым методам может потребоваться в разных ситуациях принимать разные объемы данных. Возьмем механизм, который я многократно использовал в этой книге для отображения информации. В большинстве случаев я передавал простую строку в `Console.WriteLine`, но в некоторых случаях мне требовалось форматировать и отображать другие единицы информации. Как показано в листинге 3.57, в строки можно встраивать выражения.

Листинг 3.57. Строковая интерполяция

```
Console.WriteLine($"PI: {Math.PI}. Square root of 2: {Math.Sqrt(2)}");
Console.WriteLine($"It is currently {DateTime.Now}");
var r = new Random();
Console.WriteLine(
    $"{r.Next(10)}, {r.Next(10)}, {r.Next(10)}, {r.Next(10)}");
```

Как вы помните из главы 2, когда символ \$ помещается перед строковой константой, компилятор преобразует его в вызов метода `string.Format` (функция, известная как интерполяция строк) и заменяет вложенные выражения заполнителями, такими как `{0}` и `{1}`, которые ссылаются на первый и второй аргументы после строки. Это как если бы мы написали код, показанный в листинге 3.58.

Если вы заглянете в документацию по `string.Format`, то увидите, что метод предлагает несколько перегрузок, принимающих различное количество аргументов. Очевидно, что у него конечное число перегрузок, но если вы внимательно посмотрите, то обнаружите, что, несмотря на это, метод не содержит ограничений по количеству параметров. После строки вы можете передать столько угодно аргументов, и числа в заполнителях могут быть настолько большими, насколько это необходимо для ссылок на эти аргументы. Последняя строка листинга 3.58 передает четыре аргумента после строки, и хотя класс `string` не определяет перегрузку со столькими аргументами, все работает.

Листинг 3.58. Форматирование строки

```
Console.WriteLine(string.Format(
    "PI: {0}. Square root of 2: {1}", Math.PI, Math.Sqrt(2)));
Console.WriteLine(string.Format("It is currently {0}", DateTime.Now));
var r = new Random();
Console.WriteLine(string.Format(
    "{0}, {1}, {2}, {3}",
    r.Next(10), r.Next(10), r.Next(10), r.Next(10)));
```

Как только вы передаете больше определенного количества аргументов после строки (обычно больше трех), в дело вступает одна конкретная перегрузка `string.Format`, которая принимает только два аргумента: `string` и массив `object[]`. Код, который создает компилятор для вызова метода, создает массив для хранения всех аргументов после строки и передает именно его. Таким образом, последняя строка листинга 3.58 фактически эквивалентна коду в листинге 3.59. (В главе 5 описываются массивы.)

Листинг 3.59. Явная передача нескольких аргументов в виде массива

```
Console.WriteLine(string.Format(
    "{0}, {1}, {2}, {3}",
    new object[] { r.Next(10), r.Next(10), r.Next(10) }));
```

Компилятор проделает это только с параметрами, помеченными ключевым словом `params`. Листинг 3.60 показывает, как выглядит соответствующее объявление метода `string.Format`.

Листинг 3.60. Ключевое слово `params`

```
public static string Format(string format, params object[] args)
```

Ключевое слово `params` может применяться только к последнему параметру метода, и тип этого параметра должен быть массивом. В данном случае это `object[]`, что означает, что мы можем передавать объекты любого типа, но вы можете ограничить диапазон того, что может быть передано.



Когда метод перегружен, компилятор C# ищет метод, параметры которого лучше всего соответствуют предоставленным аргументам. Он разрешит использование метода с аргументом `params`, только если более точное совпадение недоступно.

Вы можете задаться вопросом, почему класс `string` вообще предлагает перегрузки, которые принимают один, два или три аргумента типа `object`. Наличие версий с `params` как будто делает их избыточными, ведь она позволяет передавать любое количество аргументов после строки. Так какой же смысл в перегрузках, которые принимают определенное количество аргументов? Эти перегрузки существуют ради возможности избежать выделения массива. Это не значит, что использовать массивы особенно накладно; они обходятся не дороже, чем любой другой объект такого же размера. Однако выделение памяти уже достаточно затратно. Каждый объект, который вы размещаете в памяти, в конечном итоге должен быть освобожден сборщиком мусора (за исключением объектов, которые хранятся в течение всей жизни программы), поэтому уменьшение количества выделений памяти обычно хорошо влияет на производительность. Из-за этого большинство API в библиотеке классов .NET, которые принимают переменное число аргументов через `params`, тоже предлагают перегрузки, позволяющие передавать небольшое количество аргументов без необходимости выделения массива для их хранения.

Локальные функции

Вы можете определять методы внутри других методов. Они называются *локальными функциями*, и листинг 3.61 определяет две такие функции. (Вы также можете поместить их в другие подобные методы функции, такие как конструкторы или методы доступа к свойствам.)

Одно из преимуществ использования локальных функций заключается в том, что они способны облегчить чтение кода посредством перемещения

его фрагментов в именованные методы. Легче понять, что происходит, когда имеется вызов метода `GetDistance`, чем если просто выполняются вычисления внутри строки. Помните, что у этого механизма могут быть издержки, хотя в этом конкретном примере, когда я запускаю сборку `Release` на .NET Core 3.0, JIT-компилятор оказывается достаточно умен, чтобы сделать оба локальных вызова встроенными. Поэтому две локальные функции исчезают и `GetAverageDistanceFrom` становится единым методом. Таким образом, здесь мы ничего не потеряли, но с более сложными вложенными функциями JIT-компилятор может решить не использовать встраиваемые функции. И когда подобное происходит, всегда полезно знать, как компилятор C# позволяет этому коду работать.

Листинг 3.61. Локальные функции

```
static double GetAverageDistanceFrom(
    (double X, double Y) referencePoint,
    (double X, double Y)[] points)
{
    double total = 0;
    for (int i = 0; i < points.Length; ++i)
    {
        total += GetDistanceFromReference(points[i]);
    }
    return total / points.Length;

    double GetDistanceFromReference((double X, double Y) p)
    {
        return GetDistance(p, referencePoint);
    }

    static double GetDistance((double X, double Y) p1, (double X,
                                                       double Y) p2)
    {
        double dx = p1.X - p2.X;
        double dy = p1.Y - p2.Y;
        return Math.Sqrt(dx * dx + dy * dy);
    }
}
```

Метод `GetDistanceFromReference` здесь принимает один аргумент-кортеж, но использует переменную `referencePoint`, определенную его содержащим методом. Чтобы это сработало, компилятор C# перемещает эту переменную в генерированную структуру, которую он передает по ссылке методу `GetDistanceFromReference` в качестве скрытого аргумента. Вот так одна

локальная переменная может стать доступна обоим методам. Поскольку в данном примере эта сгенерированная структура передается по ссылке, переменная `referencePoint` все еще может оставаться в стеке. Однако, если вы получите делегат, ссылающийся на локальный метод, любые переменные, совместно используемые этим способом, должны будут переместиться в класс, который находится в куче, а она уже обрабатывается сборщиком мусора, что влечет за собой более высокие издержки. (См. главы 7 и 9 для более подробной информации.) Если вы желаете избежать подобных дополнительных расходов, всегда можно не предоставлять одновременный доступ к переменным внутренним и внешним методам. Начиная с C# 8.0, вы можете сообщать компилятору о своем намерении, применяя ключевое слово `static` к локальной функции, как листинг 3.61 проделывает с `GetDistance`. Это приводит к тому, что компилятор выдает ошибку, когда метод пытается использовать переменную содержащего его метода.

Помимо способа разделения методов для удобства чтения локальные функции иногда используются для обхода некоторых ограничений итераторов (см. главу 5) и асинхронных методов (глава 17). Такие методы могут возвращать управление на полпути, а затем продолжать работу позже. Это означает, что компилятор должен организовать хранение всех их локальных переменных в объекте в куче, чтобы эти переменные могли существовать столько, сколько потребуется. Это препятствует таким методам использовать определенные типы, например `Span<T>`, описанный в главе 18. В случаях, когда вам нужно использовать как `async`, так и `Span<T>`, код, который использует последнее, перемещается в неасинхронную функцию, которая находится внутри асинхронной. Это позволяет локальной функции использовать локальные переменные даже с этими неудобными типами.

Методы с телом-выражением

Если вы пишете метод, содержащий лишь единственный оператор `return`, можно использовать более лаконичный синтаксис. В листинге 3.62 показан альтернативный способ написания метода `GetDistanceFromReference` из раздела «Локальные функции» на с. 209. (Как вы, наверное, заметили, я уже использовал его в нескольких других примерах.)

Листинг 3.62. Метод с телом-выражением

```
double GetDistanceFromReference((double X, double Y) p)
    => GetDistance(p, referencePoint);
```

Вместо тела метода вы пишете =>, за которым следует выражение, которое в противном случае следовало бы после ключевого слова `return`. Этот синтаксис основан на лямбда-синтаксисе, который вы можете использовать для написания встроенных функций и построения деревьев выражений. Они обсуждаются в главе 9.

Методы расширения

C# позволяет вам писать методы, которые выглядят как новые члены существующих типов. *Методы расширения*, а именно так они называются, выглядят как обычные статические методы, но с ключевым словом `this`, добавленным перед первым параметром. Вам разрешено определять методы расширения только в статическом классе. Листинг 3.63 добавляет один не особо полезный метод расширения с именем `Show` к `string`.

Листинг 3.63. Метод расширения

```
namespace MyApplication
{
    public static class StringExtensions
    {
        public static void Show(this string s)
        {
            System.Console.WriteLine(s);
        }
    }
}
```

Я показал в этом примере объявление пространства имен, потому что они здесь имеют большое значение: методы расширения доступны, только когда вы написали директиву `using` для пространства имен, в котором определено расширение, или если код, который вы пишете, определен в том же пространстве имен. В коде, который не выполняет никакое из этих условий, класс `string` будет выглядеть как обычно и не получит метод `Show`, показанный в листинге 3.63.

Однако код листинга 3.64, который определен в том же пространстве имен, что и метод расширения, увидит, что данный метод доступен.

Код в листинге 3.65 находится в другом пространстве имен, но также имеет доступ к методу расширения благодаря директиве `using`.

Листинг 3.64. Метод расширения доступен в силу объявления пространства имен

```
namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            "Hello".Show();
        }
    }
}
```

Листинг 3.65. Метод расширения доступен благодаря директиве using

```
using MyApplication;
```

```
namespace Other
{
    class Program
    {
        static void Main(string[] args)
        {
            "Hello".Show();
        }
    }
}
```

Методы расширения на самом деле не являются членами класса, для которого они определены. Класс `string` фактически не получает в этих примерах дополнительного метода — это иллюзия, создаваемая компилятором C#, которую он поддерживает даже в ситуациях, когда вызов метода происходит неявно. Это особенно полезно с функциями C#, которые требуют наличия определенных методов. В главе 2 вы увидели, что циклы `foreach` зависят от метода `GetEnumerator`. Многие функции LINQ, которые мы рассмотрим в главе 10, также зависят от наличия определенных методов, так же как и функции асинхронного языка, описанные в главе 17. Во всех этих случаях вы можете подключить указанный функционал для типов, которые не поддерживают его напрямую, написав подходящие методы расширения.

Свойства

Классы и структуры могут определять свойства, которые по сути являются просто скрытыми методами. Для доступа к свойству вы используете синтак-

сис, который выглядит как доступ к полю, но в итоге вызывает метод. Свойства могут быть полезны для обозначения намерения. Когда нечто представлено как свойство, подразумевается, что оно содержит информацию об объекте, а не операцию, которую выполняет объект. Ввиду этого чтение свойства обычно не затратно и не должно иметь заметных побочных эффектов. Методы, с другой стороны, с большой вероятностью заставляют объект что-то делать.

Конечно, поскольку свойства — это всего лишь методы, ничто не обязывает вас следовать этому правилу. Вы можете создать свойство, которое работает часами и вносит значительные изменения в состояние вашего приложения всякий раз, когда его значение читается, но это будет довольно паршивым способом писать код.

Свойства обычно предоставляют пару методов: один для получения значения и один для его установки.

В листинге 3.66 показана очень распространенная схема: свойство с методами `get` и `set`, которые предоставляют доступ к полю. Почему бы просто не сделать поле публичным? Это не одобряется, потому что позволяет коду извне менять состояние объекта, при этом не ставя его в известность об этом. Может статься, что в будущих версиях кода объект должен будет что-то делать — например, обновлять пользовательский интерфейс — каждый раз, когда значение изменяется. В любом случае, поскольку свойства содержат код, они предлагают больше гибкости, чем поля с доступом `public`. Например, вы можете сохранить данные в другом формате, отличном от того, который возвращает свойство, или даже реализовать свойство, которое вычисляет их значение из других свойств. Другая причина использования свойств заключается в том, что это требуется некоторыми системами. Например, некоторые системы привязки данных пользовательского интерфейса умеют только получать свойства. Кроме того, некоторые типы не поддерживают поля экземпляров. Позже в этой главе я покажу, как определять абстрактный тип с помощью ключевого слова `interface`, а интерфейсы могут содержать свойства, но не поля экземпляра.



Внутри метода доступа `set` особое значение имеет `value`. Это *контекстное ключевое слово* — текст, который в определенных контекстах воспринимается языком как ключевое. Вне свойства вы можете использовать `value` в качестве идентификатора, но внутри свойства оно представляет значение, которое вызывающий объект хочет присвоить свойству.

Листинг 3.66. Класс с простым свойством

```
public class HasProperty
{
    private int _x;
    public int X
    {
        get
        {
            return _x;
        }
        set
        {
            _x = value;
        }
    }
}
```

В случаях, когда тело `get` содержит лишь оператор `return` или когда `set` — оператор с одним выражением, вы можете использовать синтаксис члена с телом-выражением, показанный в листинге 3.67. (Очень похоже на синтаксис метода, показанный в листинге 3.62.)

Листинг 3.67. Set и get с телом-выражением

```
public class HasProperty
{
    private int _x;
    public int X
    {
        get => _x;
        set => _x = value;
    }
}
```

Схема из листингов 3.66 и 3.67 настолько распространена, что C# может сам написать большую ее часть за вас. Листинг 3.68 более или менее равнозначен — компилятор генерирует для нас поле и создает методы `get` и `set`, которые извлекают и изменяют значение так же, как и в листинге 3.66. Единственное отличие состоит в том, что код, расположенный в другом месте того же класса, не может попасть непосредственно в поле в листинге 3.68, потому что компилятор его скрывает. Официальное имя в спецификации языка — автоматически реализуемое свойство, но обычно их называют просто автосвойствами.

Листинг 3.68. Автосвойство

```
public class HasProperty
{
    public int X { get; set; }
}
```

Используете ли вы явные или автоматические свойства, это всего лишь причудливый синтаксис для пары методов. Метод `get` возвращает значение объявленного типа свойства — в данном случае `int`, — в то время как метод `set` принимает один аргумент этого типа через скрытый параметр `value`. Листинг 3.66 использует этот аргумент для обновления поля, но вы, конечно, не обязаны сохранять это значение в поле. Фактически ничто даже не заставляет вас каким-либо образом связывать методы `get` и `set` — вы можете написать метод `get`, который возвращает случайные значения, и метод `set`, который полностью игнорирует значение, которое вы в него передаете. Однако то, что вы *можете*, не означает, что вам это *нужно*. На практике любой, кто использует ваш класс, будет ожидать, что свойства запомнят переданные им значения, не в последнюю очередь потому, что при использовании свойства выглядят как поля, что показано в листинге 3.69.

Листинг 3.69. Использование свойства

```
var o = new HasProperty();
o.X = 123;
o.X += 432;
Console.WriteLine(o.X);
```

Если для реализации свойства вы используете полный синтаксис, показанный в листинге 3.66, или тело-выражение из листинга 3.67, то можете опустить либо `set`, либо `get`, что сделает свойство предназначенным только для чтения или только для записи. Свойства, доступные только для чтения, могут быть полезны для тех частей объектов, которые должны оставаться неизменными все время жизни, таких как идентификаторы, или свойства, которые вычисляются на основе других свойств. Свойства только для записи менее полезны, хотя могут проскакивать в механизмах внедрения зависимостей. Вы не можете создать свойство только для записи с синтаксисом автосвойства, показанным в листинге 3.68, потому что не сможете сделать с установленным значением ничего полезного.

Существует два варианта свойств только для чтения. Иногда полезно иметь свойство только для чтения, но доступное для изменения вашим классом.

Вы можете определить свойство, где метод `get` является открытым, а метод `set` — нет (или наоборот, для свойства только для записи). Это можно проделать как с полным, так и с автоматическим синтаксисом. Листинг 3.70 показывает, как это выглядит с последним вариантом.

Листинг 3.70. Автосвойство с закрытым методом `set`

```
public int X { get; private set; }
```

Если вам требуется, чтобы свойство было доступно только для чтения в том смысле, что его значение никогда не меняется после создания, вы можете при использовании синтаксиса автосвойства совсем опустить метод `set`, как показано в листинге 3.71.

Листинг 3.71. Автосвойство без метода `set`

```
public int X { get; }
```

Вы можете задаться вопросом: как же установить значение такого свойства, если у вас нет ни метода `set`, ни прямого доступа к полю? Ответ состоит в том, что внутри конструктора вашего объекта это свойство можно устанавливать. (На самом деле если вы опустите `set`, то никакого метода установки значения не появится — компилятор лишь генерирует код, который устанавливает поле непосредственно, когда вы делаете это в конструкторе с помощью `set`.) Автосвойство только с одним лишь `get` по сути эквивалентно полю только для чтения, обернутому обычным свойством, предназначенному только для получения. Как и в случае с полями, вы можете написать инициализатор для предоставления ему начального значения. Листинг 3.72 использует оба стиля. Если вы используете конструктор, который не принимает аргументов, значение свойства будет равно 42, а если другой, то оно будет иметь любое значение, которое вы предоставите.

Иногда необходимо записать в свойство только для чтения значение, полностью вычисляемое на основе других свойств. Например, если вы написали тип, представляющий вектор со свойствами, называемыми `X` и `Y`, то вы можете добавить свойство, которое возвращает величину вектора, вычисленную из этих двух других свойств, как показано в листинге 3.73.

Есть и более компактный способ записать это. Мы могли бы использовать синтаксис тела-выражения, показанный в листинге 3.67, но в случае свойства только для чтения мы можем пойти еще дальше и поместить `=>` и выражение

сразу после имени свойства. (Это позволяет нам опустить скобки и ключевое слово `get`.) Листинг 3.74 в точности соответствует листингу 3.73.

Листинг 3.72. Инициализация автосвойства без `set`

```
public class WithAutos
{
    public int X { get; } = 42;

    public WithAutos()
    {
    }

    public WithAutos(int val)
    {
        X = val;
    }
}
```

Листинг 3.73. Вычисляемое свойство

```
public double Magnitude
{
    get
    {
        return Math.Sqrt(X * X + Y * Y);
    }
}
```

Листинг 3.74. Свойство только для чтения с телом-выражением

```
public double Magnitude => Math.Sqrt(X * X + Y * Y);
```

Говоря о свойствах только для чтения, важно знать о связанных свойствах, значимых типах и неизменяемости.

Свойства и изменяемые значимые типы

Как я упоминал ранее, значимые типы имеют тенденцию быть более простыми, если являются неизменяемыми, но это не обязательное требование. Одна из причин, по которой следует избегать изменяемых значимых типов, заключается в том, что вы можете случайно изменить копию значения, а не само значение. Эта проблема станет очевидной, когда вы определите свойство, использующее изменяемый значимый тип. Структура `Point` в пространстве имен `System.Windows` является изменяемой, поэтому мы можем

использовать ее для иллюстрации проблемы. Листинг 3.75 определяет свойство `Location` этого типа.

Тип `Point` определяет свойства для чтения/записи, называемые `X` и `Y`, поэтому, задав переменную типа `Point`, вы можете эти свойства устанавливать. Однако, если вы попытаетесь установить одно свойство через другое, код не скомпилируется. Листинг 3.76 пытается изменить свойство `X` типа `Point`, полученное из свойства `Location` объекта `Item`.

Листинг 3.75. Свойство, использующее изменяемый значимый тип
`using System.Windows;`

```
public class Item
{
    public Point Location { get; set; }
}
```

Листинг 3.76. Ошибка: невозможно изменить свойство свойства значимого типа

```
var item = new Item();
item.Location.X = 123; // Не будет компилироваться
```

В этом примере выдается следующая ошибка:

```
error CS1612: Cannot modify the return value of 'Item.Location'
because it is not a variable
```

C# рассматривает поля как переменные, так же как и в случае локальных переменных и аргументов метода, поэтому, если мы изменим листинг 3.75 так, чтобы `Location` был открытым полем, а не свойством, листинг 3.76 скомпилируется и будет работать. Но почему это не сработает со свойством? Вспомните, что свойства — это просто методы, поэтому листинг 3.75 более или менее равнозначен листингу 3.77.

Листинг 3.77. Замена свойства методами

```
using System.Windows;

public class Item
{
    private Point _location;
    public Point get_Location()
    {
        return _location;
    }
}
```

```
    }
    public void set_Location(Point value)
    {
        _location = value;
    }
}
```

Поскольку `Point` является значимым типом, `get_Location` возвращает копию. Вам может быть интересно, можно ли использовать функционал возврата `ref`, описанный ранее. Мы, конечно, могли бы так поступить с простыми методами, но у этого функционала есть несколько ограничений применительно к свойствам. Во-первых, нельзя определить автосвойство с типом `ref`. Во-вторых, нельзя определить доступное для записи свойство с типом `ref`. Но вы можете определить свойство `ref` только для чтения, как показано в листинге 3.78.

Листинг 3.78. Свойство, возвращающее ссылку

```
using System.Windows;
```

```
public class Item
{
    private Point _location;

    public ref Point Location => ref _location;
}
```

С такой реализацией `Item` код в листинге 3.76 работает нормально. (Как ни странно, чтобы сделать свойство модифицируемым, пришлось превратить его в свойство только для чтения.)

До того как возврат `ref` был добавлен в C#, не было никакого способа заставить это работать. Все возможные реализации свойства в конечном итоге возвращали бы копию значения свойства, поэтому, если компилятор разрешал компиляцию в листинге 3.76, мы бы устанавливали свойство `X` в копии, возвращаемой свойством, а не фактическое значение в объекте `Item`, которое представляет свойство. Листинг 3.79 делает это явным и действительно компилируется, так как компилятор позволяет нам пилить сук, на котором сидим, если выразить это желание достаточно ясно. И совершенно очевидно, что в этой версии кода значение в объекте `Item` не изменится.

Однако при реализации свойства в листинге 3.78 код в листинге 3.76 компилируется и в конечном итоге ведет себя, как тот, что показан в листин-

ге 3.80. Здесь мы видим, что ссылка на `Point` получена, поэтому, когда мы устанавливаем свойство `X`, мы воздействуем на то, на что ссылаемся (поле `_location` в `Item` в нашем случае), а не на локальную копию.

Листинг 3.79. Делаем копию явной

```
var item = new Item();
Point location = item.Location;
location.X = 123;
```

Листинг 3.80. Делаем ссылку явной

```
var item = new Item();
ref Point location = ref item.Location;
location.X = 123;
```

Так что это стало возможным благодаря довольно недавним дополнениям к языку. Но тут тоже легко ошибиться. К счастью, большинство значимых типов являются неизменяемыми, а эта проблема возникает только с изменяемыми значимыми типами.



Неизменяемость не решает проблему наверняка — вы все еще не всегда можете писать код, который вам нужен, например `item.Location.X = 123`. Но, по крайней мере, неизменяемые структуры не вводят вас в заблуждение, создавая впечатление, что их можно менять.

Поскольку все свойства по сути являются просто методами (обычно парными), теоретически они могут принимать аргументы кроме неявного аргумента `value`, используемого методами `set`. CLR позволяет это, но C# не поддерживает подобное, за исключением одного особого вида свойства: индексатора.

Индексаторы

Индексатор — это свойство, которое принимает один или несколько аргументов, а доступ к нему осуществляется с использованием того же синтаксиса, что и для массивов. Это полезно, когда вы пишете класс, содержащий коллекцию объектов. В листинге 3.81 используется один из классов коллекций, предоставляемый библиотекой классов .NET, а именно `List<T>`. По сути, это массив переменной длины, и он выглядит как встроенный массив благодаря

своему индексатору во второй и третьей строках. (Я подробно опишу массивы и типы коллекций в главе 5. И я опишу обобщенные типы, к которым относится `List<T>`, в главе 4.)

С точки зрения CLR индексатор — это свойство, очень похожее на любое другое, за исключением того, что оно определяется как свойство по умолчанию. Эта концепция является пережитком старых версий Visual Basic на основе COM, который был перенесен в .NET и который C# в основном игнорирует. Индексаторы — единственная функция C#, которая рассматривает свойства по умолчанию как особые. Если класс определяет свойство как свойство по умолчанию и если свойство принимает хотя бы один аргумент, C# позволяет вам использовать это свойство через синтаксис индексатора.

Листинг 3.81. Использование индексатора

```
var numbers = new List<int> { 1, 2, 1, 4 };
numbers[2] += numbers[1];
Console.WriteLine(numbers[0]);
```

Синтаксис объявления индексаторов несколько своеобразен. В листинге 3.82 показан индексатор только для чтения. Можно добавить доступ для чтения/записи посредством `set`, как и для любого другого свойства⁶.

Листинг 3.82. Класс с индексатором

```
public class Indexed
{
    public string this[int index]
    {
        get => index < 5 ? "Foo" : "bar";
    }
}
```

C# поддерживает и многомерные индексаторы. Это просто индексаторы с более чем одним параметром. Поскольку свойства на самом деле яв-

⁶ Кстати, свойство по умолчанию имеет имя, потому что это требуется от всех свойств. C# вызывает свойство индексатора `Item` и автоматически добавляет аннотацию, указывающую, что это свойство по умолчанию. Обычно вам не нужно ссылаться на индексатор по имени, но имя будет видно в некоторых инструментах. В документации библиотеки классов .NET перечислены индексаторы `Item`, хотя это имя редко используется в коде.

ляются методами, вы можете определить индексаторы с любым количеством параметров. Можно использовать любую комбинацию типов для параметров.

Как вы помните из главы 2, C# содержит null-условные операторы. В той главе мы видели, что они используются для доступа к свойствам и полям. Например, `myString?.Length` будет иметь тип `int?`, а его значение будет `null`, если `myString` равно `null` и содержать значением свойства `Length` в ином случае. Существует еще одна форма null-условного оператора, которую можно использовать с индексатором, как показано в листинге 3.83.

Листинг 3.83. Доступ с помощью null-условного индекса

```
string? s = objectWithIndexer?[2];
```

Как и в случае null-условного поля или доступа к свойству, здесь генерируется код, который проверяет, является ли левая часть (в данном случае `objectWithIndexer`) равной `null`. Если это так, все выражение вычисляется как `null`; индексатор же вызывается, только если левая часть выражения не равна `null`. По сути, это эквивалентно коду, показанному в листинге 3.84.

Листинг 3.84. Код, эквивалентный null-условному доступу по индексу

```
string? s = objectWithIndexer == null ? null : objectWithIndexer[2];
```

Этот синтаксис null-условного индекса работает и с массивами.

Синтаксис инициализатора

Часто при создании объекта вам захочется установить определенные свойства, потому что может оказаться невозможным предоставить всю необходимую информацию через аргументы конструктора. Это особенно характерно для объектов, которые представляют собой настройки для управления некоторыми операциями. Например, тип `ProcessStartInfo` позволяет настраивать множество различных аспектов только что созданного процесса ОС. Он имеет 16 свойств, но вам, как правило, нужно установить только несколько из них в любом конкретном сценарии. Даже если вы предполагаете, что имя файла для запуска должно присутствовать всегда, по-прежнему остается 32 768 возможных комбинаций свойств. Вы точно не хотели бы иметь конструктор для каждого из них.

На практике класс может предлагать конструкторы для нескольких наиболее распространенных комбинаций, но для всего остального вы просто устанавливаете свойства после создания. C# предлагает краткий способ создания объекта с установкой некоторых его свойств в одном выражении. Листинг 3.85 использует этот синтаксис инициализатора объекта. Он работает и с полями, хотя наличие публичных полей для записи достаточно необычно.

Листинг 3.85. Использование инициализатора объекта

```
Process.Start(new ProcessStartInfo
{
    FileName = "cmd.exe",
    UseShellExecute = true,
    WindowStyle = ProcessWindowStyle.Maximized,
});
```

Вы также можете указать и аргументы конструктора. Листинг 3.86 работает так же, как и листинг 3.85, но передает имя файла в качестве аргумента конструктора (потому что это одно из немногих свойств, которые ProcessStartInfo позволяет вам подобным образом указывать).

Листинг 3.86. Использование конструктора и инициализатора объекта

```
Process.Start(new ProcessStartInfo("cmd.exe")
{
    UseShellExecute = true,
    WindowStyle = ProcessWindowStyle.Maximized,
});
```

Синтаксис инициализатора объекта может избавить вас от необходимости в отдельной переменной для ссылки на объект на время установки нужных свойств. Как показывают листинги 3.85 и 3.86, вы можете передать объект, инициализированный таким образом, непосредственно в качестве аргумента метода. Важным итогом этого является то, что инициализация такого рода может содержаться целиком в одном выражении. Это важно в сценариях, использующих деревья выражений, которые мы рассмотрим в главе 9.

В этом синтаксисе есть вариант, который позволяет передавать значения в индексатор в инициализаторе объекта. Словарь в листинге 3.87 инициализируется именно таким образом. (Глава 5 подробно описывает словари и другие типы коллекций.)

Листинг 3.87. Использование индексатора в инициализаторе объекта

```
var d = new Dictionary<string, int>
{
    ["One"] = 1,
    ["Two"] = 2,
    ["Three"] = 3
};
```

Операторы

Классы и структуры могут определять пользовательские значения операторов. Ранее я показывал некоторые пользовательские операторы: листинг 3.20 давал определения для `==` и `!=`. Класс или структура может поддерживать почти все арифметические, логические и операторы сравнения, показанные в главе 2. Из операторов, показанных в табл. 2.3–2.6, можно определить пользовательские значения для всех, кроме операторов условного `И` (`&&`) и условного `ИЛИ` (`||`). Эти операторы вычисляются на основе других операторов, поэтому, определив логическое `И` (`&`), логическое `OR` (`|`), а также логические `true` и `false` (кратко описанные ниже), вы можете управлять тем, как `&&` и `||` работают для вашего типа, даже если не можете реализовать их напрямую.

Все пользовательские реализации операторов следуют определенной схеме. Они выглядят как статические методы, но там, где вы обычно ожидаете имя метода, расположено ключевое слово `operator`, за ним следует сам оператор, для которого вы хотите определить пользовательское значение. После этого следует список параметров, где количество параметров определяется количеством operandов, которые требует оператор.

Листинг 3.88 показывает, как бинарный оператор `+` будет выглядеть для класса `Counter`, определенного ранее в этой главе.

Листинг 3.88. Реализация оператора `+`

```
public static Counter operator +(Counter x, Counter y)
{
    return new Counter { _count = x._count + y._count };
}
```

Хотя количество аргументов должно соответствовать количеству operandов, необходимых оператору, только один из аргументов обязан совпадать

с определяющим типом. Листинг 3.89 использует эту особенность, чтобы позволить прибавлять класс Counter к int.

C# требует, чтобы некоторые операторы были определены попарно. Мы уже наблюдали это в случае с операторами == и !=, которые нельзя определять по одному. Аналогично, если вы определяете оператор > для своего типа, вы должны определить и оператор <, и наоборот. То же самое будет верным и для пары >= и <=. (Есть еще одна пара, операторы true и false, но они немного отличаются; я скоро к ним вернусь.)

Листинг 3.89. Поддержка других типов operandов

```
public static Counter operator +(Counter x, int y)
{
    return new Counter { _count = x._count + y };
}

public static Counter operator +(int x, Counter y)
{
    return new Counter { _count = x + y._count };
}
```

Когда вы перегружаете оператор, для которого имеется составной оператор присваивания, вы фактически определяете поведение обоих. Например, если вы определяете пользовательское поведение для оператора +, оператор += также будет работать.

Ключевое слово `operator` может определять и пользовательские преобразования — методы, которые преобразуют ваш тип в какой-либо другой тип или из него. Например, если мы хотим преобразовать объекты Counter в int и из int, то можем добавить в класс два метода из листинга 3.90.

Листинг 3.90. Операторы преобразования

```
public static explicit operator int(Counter value)
{
    return value._count;
}

public static explicit operator Counter(int value)
{
    return new Counter { _count = value };
}
```

Я использовал явное ключевое слово `explicit`, которое означает, что эти преобразования доступны с использованием синтаксиса приведения, как показано в листинге 3.91.

Листинг 3.91. Использование операторов явного преобразования

```
var c = (Counter) 123;
var v = (int) c;
```

Если вы используете ключевое слово `implicit` вместо `explicit`, ваше преобразование сможет произойти без необходимого приведения. В главе 2 мы увидели, что некоторые преобразования происходят неявно: в определенных ситуациях C# автоматически повышает числовые типы. Например, вы можете использовать `int` там, где ожидается `long`, например в качестве аргумента для метода или в присваивании. Преобразование из `int` в `long` всегда будет успешным и никогда не потеряет информацию, поэтому компилятор автоматически генерирует код для выполнения такого преобразования, не требуя явного приведения. Если вы пишете неявные операторы преобразования, компилятор C# будет точно так же молча их применять, позволяя использовать ваш пользовательский тип там, где ожидается какой-то другой. (На самом деле спецификация C# позволяет производить числовые повышения, такие как преобразование из `int` в `long`, как встроенные неявные преобразования.)

Операторы неявного преобразования не пишут часто. Это следует делать только в том случае, если вы сможете соответствовать тем же стандартам, что применяются к встроенным повышениям: преобразование всегда должно быть возможным и никогда не должно вызывать исключения. Более того, преобразование не должно преподносить сюрпризы, так как неявные преобразования — это довольно хитрая штука в том смысле, что они позволяют вызывать методы с помощью кода, который не похож на вызов метода. Поэтому, если вы не планируете вводить в заблуждение других разработчиков, используйте неявные преобразования только в тех случаях, в которых их можно трактовать однозначно.

C# распознает еще два оператора: `true` и `false`. Если вы определите любой из них, вы должны определить и второй. Это немного странная пара, потому что спецификация C# определяет их как перегрузки унарных операторов, но они не соответствуют напрямую ни одному оператору, который вы можете написать в выражении. Они вступают в дело при двух сценариях.

Если вы не определили неявное преобразование в `bool`, но определили операторы `true` и `false`, C# будет использовать оператор `true`, если вы используете свой тип в качестве выражения для оператора `if`, цикла `do` или `while` или как выражение условия в цикле `for`. Однако компилятор предполагает неявный оператор `bool`, так что это не главная причина существования операторов `true` и `false`.

Основной сценарий использования операторов `true` и `false` состоит в том, чтобы позволить вашему пользовательскому типу выступать в качестве операнда условного логического оператора (`&&` или `||`). Помните, что эти операторы будут вычислять свой второй operand, только если первый результат не полностью определяет итог. Если вы хотите настроить поведение этих операторов, вы не можете сделать это напрямую. Вместо этого вы должны определить безусловные версии операторов (`&` и `|`), а также определить операторы `true` и `false`. При вычислении `&&` C# будет использовать ваш оператор `false` для первого операнда, и если он укажет на то, что первый operand является ложным, он будет утверждать себя вычислением второго. Если первый operand не ложный, то будет вычислен второй operand, а затем оба они будут переданы в ваш пользовательский оператор `&`. Оператор `||` работает во многом так же, но с операторами `true` и `|` соответственно.



Операторы `true` и `false` присутствовали еще в первой версии C#, и их основное применение состояло в том, чтобы добавить реализацию типов, поддерживающих логические значения, допускающие значение `NULL`, с семантикой, аналогичной той, что предлагается во многих базах данных. Поддержка типов, допускающих значение `NULL`, добавленная в C# 2.0, предлагает для этого лучшее решение, поэтому указанные операторы больше не являются особенно полезными. Тем не менее все еще есть некоторые старые части библиотеки классов .NET, которые от них зависят.

Мы можете спросить, а зачем вообще нужны специальные операторы `true` и `false`? Неужели нельзя просто определить неявное преобразование в `bool`? Фактически это можно сделать, и если мы так поступим вместо предоставления `&`, `|`, `true` и `false`, C# использует это определение для реализации `&&` и `||` для нашего типа. Однако некоторые типы могут представлять значения, которые не являются ни `true`, ни `false`, а представляют неизвестное третье состояние. Оператор `true` позволяет C# задаваться вопросом «действительно ли это `true`?», а объект может отвечать «нет», не

подразумевая при этом, что что-то определенно ложно. Преобразование в `bool` этого не поддерживает.

Никакие другие операторы не могут быть перегружены. Например, вы не можете определить пользовательские значения оператора доступа к членам метода, условного оператора (`? :`), оператора объединения с неопределенным значением (`??`) или оператора `new`.

События

Структуры и классы могут объявлять события. Этот тип члена позволяет типу выдавать уведомления, когда происходит что-то интересующее нас, используя при этом модель на основе подписки. Например, объект пользовательского интерфейса, представляющий кнопку, может определять событие `Click`, а вы можете написать код, который подписывается на это событие.

События зависят от делегатов, и сейчас я не буду вдаваться в подробности — поговорим об этом в главе 9. Я упоминаю их только потому, что без них этот раздел о членах типа был бы неполным.

Вложенные типы

Последний тип члена, который мы можем определить в классе или структуре, — это вложенный тип. Вы можете определить вложенные классы, структуры или любые другие типы, описанные далее в этой главе. Вложенный тип умеет все, что делает его обычный аналог, но при этом обладает рядом дополнительных особенностей.

Когда тип становится вложенным, у вас появляется больше вариантов видимости. Тип, определенный в глобальной области видимости, может быть только `public` или `internal` — тип `private` не имеет смысла, потому что делает что-то доступным только внутри содержащего типа, но если вы определяете что-то в глобальной области, у вас нет содержащего типа. Но у вложенного типа есть содержащий тип, поэтому, если вы определяете вложенный тип и делаете его `private`, он может использоваться только внутри типа, в который он вложен. Листинг 3.92 показывает закрытый класс.

Закрытые классы могут быть полезны в ситуациях, когда вы используете API, который требует реализации определенного интерфейса. В данном случае я вызываю `Array.Sort` для сортировки списка файлов по длине их

имен. (Это не слишком полезно, но выглядит красиво.) Я предоставляю пользовательский порядок сортировки в форме объекта, который реализует интерфейс `IComparer<string>`. Я подробно опишу интерфейсы в следующем разделе, но этот интерфейс — всего лишь описание того, что от нас требуется методу `Array.Sort`. Для реализации этого интерфейса я написал собственный класс. Но этот класс — просто деталь реализации остальной части моего кода, поэтому я не хочу делать его публичным. Вложенный класс `private` — это как раз то, что мне нужно.

Листинг 3.92. Закрытый вложенный класс

```
class Program
{
    private static void Main(string[] args)
    {
        // Запрашиваем библиотеку классов о том, где находится папка
        // пользователя Мои Документы
        string path =
            Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments);
        string[] files = Directory.GetFiles(path);
        var comparer = new LengthComparer();
        Array.Sort(files, comparer);
        foreach (string file in files)
        {
            Console.WriteLine(file);
        }
    }

    private class LengthComparer : IComparer<string>
    {
        public int Compare(string x, string y)
        {
            int diff = x.Length - y.Length;
            return diff == 0 ? x.CompareTo(y) : diff;
        }
    }
}
```

Коду во вложенном типе разрешено использовать непубличные члены содержащего типа. Однако экземпляр вложенного типа автоматически не получает ссылку на экземпляр содержащего типа. (Если вы знакомы с Java, это может вас удивить. Вложенные классы C# эквивалентны статическим вложенным классам Java, а там нет эквивалента внутреннему классу.) Если

вам нужно, чтобы вложенные экземпляры имели ссылку на свой контейнер, вам нужно объявить поле для ее хранения и озаботиться ее инициализацией. Работать он будет точно так же, как любой объект, который хочет содержать ссылку на другой. Очевидно, это сработает, только если внешний тип является ссылочным.

До сих пор мы рассматривали только классы и структуры, но в C# есть несколько других способов определения пользовательских типов. Некоторые из них достаточно сложны, чтобы получить по собственной главе, но есть и несколько простых, о которых я расскажу здесь.

Интерфейсы

Интерфейс определяет интерфейс программирования. Очень часто интерфейсы полностью лишены реализации, но C# 8.0 добавляет возможность определять реализации по умолчанию для некоторых или всех методов, а также определять вложенные типы и статические поля. (Интерфейсы не могут определять нестатические поля.) Классы могут реализовывать интерфейсы по собственному выбору. Если вы напишите код, который работает в рамках интерфейса, он сможет работать со всем, что реализует этот интерфейс, вместо того чтобы ограничиваться работой с одним конкретным типом.

Например, библиотека классов .NET включает интерфейс `IEnumerable<T>`, который определяет минимальный набор членов для представления последовательностей значений. (Это общий интерфейс, поэтому он может представлять последовательности чего угодно. Скажем, `IEnumerable<string>` – это последовательность строк. Обобщения обсуждаются в главе 4.) Если метод имеет параметр типа `IEnumerable <string>`, вы можете передать ему ссылку на экземпляр любого типа, который реализует этот интерфейс, что означает, что один метод может работать с массивами, различными классами коллекций, предоставляемыми библиотекой классов .NET, некоторыми функциями LINQ и многими другими.

Интерфейс объявляет методы, свойства и события, но он не может определять их тела, как показано в листинге 3.93. Свойства указывают на то, должны ли присутствовать операторы `set` или `get`, но в нашем случае вместо тел операторов мы видим точки с запятой. Интерфейс – это фактически список членов, которые тип должен предоставить, если он хочет реализовать интерфейс. До C# 8.0 эти подобные методам члены были единственными

типами элементов, которые могли содержаться в интерфейсах. Я еще расскажу о дополнительных типах элементов, доступных в настоящее время, но большинство интерфейсов, с которыми вы, скорее всего, на сегодняшний день встретитесь, содержат только такие типы элементов.

Отдельные члены, подобные методам, не допускают модификаторов доступа — их видимость контролируется на уровне самого интерфейса. (Как и классы, интерфейсы являются либо `public`, либо `internal`, если только они не являются вложенными, и в этом случае могут иметь любую видимость.) Интерфейсы не могут объявлять конструкторы, а могут только сообщить, какие сервисы должен предоставить объект после его конструирования.

Листинг 3.93. Интерфейс

```
public interface IDoStuff
{
    string this[int i] { get; set; }
    string Name { get; set; }
    int Id { get; }
    int SomeMethod(string arg);
    event EventHandler Click;
}
```

Кстати, большинство интерфейсов в .NET следуют соглашению, что их имя начинается с заглавной буквы `I`, за которой следует одно или несколько слов в `PascalCasing`.

Класс объявляет реализуемые интерфейсы в списке после двоеточия, следующего за именем класса, как показано в листинге 3.94. Он должен содержать реализации всех членов, перечисленных в интерфейсе. Если хотя бы одного нет, то вы получите ошибку компилятора.

Листинг 3.94. Реализация интерфейса

```
public class DoStuff : IDoStuff
{
    public string this[int i] { get { return i.ToString(); } set { } }
    public string Name { get; set; }
    ...etc
}
```

Когда мы реализуем интерфейс в C#, мы обычно определяем каждый из его методов как открытый член нашего класса. Однако иногда вам может потребоваться этого избежать. Иногда некоторые API могут требовать от

вас реализации интерфейса, который, по вашему мнению, нарушает чистоту API вашего класса. Или, что чаще встречается, вы уже определили члена с теми же именем и сигнатурой, которые требует интерфейс, но делает он что-то другое. Или, что еще хуже, вам может понадобиться реализовать два разных интерфейса, каждый из которых определяет элементы, которые имеют одинаковые имена и сигнатурьы, но требуют различного поведения. Вы можете решить любую из этих проблем с помощью метода, называемого явной реализацией, для определения членов, которые реализуют член интерфейса, не будучи при этом публичными. Листинг 3.95 показывает такой синтаксис для реализации одного из методов интерфейса в листинге 3.93. В явных реализациях вы не указываете видимость и добавляете имя интерфейса перед именем члена.

Листинг 3.95. Явная реализация члена интерфейса

```
int IDoStuff.SomeMethod(string arg)
{
    ...
}
```

Когда тип использует явную реализацию интерфейса, такие члены не могут использоваться через ссылку на сам тип. Они становятся видимыми только при обращении к объекту через тип интерфейса.

Когда класс реализует интерфейс, он становится неявно конвертируемым в этот тип интерфейса. Таким образом, вы можете передать любое выражение типа `DoStuff` из листинга 3.94 в качестве аргумента метода типа `IDoStuff`.

Интерфейсы являются ссылочными типами. Несмотря на это, вы можете реализовывать интерфейсы как для классов, так и для структур. Тем не менее вы должны быть осторожны, когда делаете это со структурой, потому что когда вы получите ссылку на структуру, реализующую интерфейс, это будет ссылка на упаковку, которая фактически является объектом, содержащим копию структуры так, что к нему можно обращаться через ссылку. Мы рассмотрим упаковку в главе 7.

Реализация интерфейса по умолчанию

Новая функция в C# 8.0, называемая реализацией интерфейса по умолчанию, позволяет включать некоторые детали реализации в определение интерфейса. Функционал зависит от поддержки средой выполнения, поэтому

это доступно только в коде, предназначенному для .NET Core 3.0 или более поздней версии или .NET Standard 2.1 или более поздней версии. Вы можете предоставить статические поля, вложенные типы и тела для методов, методы доступа к свойствам и методы `add` и `remove` для событий (которые я опишу в главе 9). Листинг 3.96 показывает, как можно реализовать определение свойства по умолчанию.

Листинг 3.96. Интерфейс с реализацией свойства по умолчанию

```
public interface INamed
{
    int Id { get; }
    string Name => $"{this.GetType():t}: {this.Id}";
}
```

Если класс решит реализовать `INamed`, то ему потребуется только предоставить реализацию для свойства `Id` этого интерфейса. Он также может предоставить свойство `Name`, если хочет, но уже не обязательно. Если класс не определяет собственное свойство `Name`, вместо него будет использоваться определение из интерфейса.

Реализации интерфейса по умолчанию обеспечивают частичное решение давнего ограничения интерфейсов: если вы определяете интерфейс, который затем делаете доступным для другого кода (например, через библиотеку классов), добавление новых членов в этот интерфейс может вызвать проблемы в существующем коде, который его использует. Код, который вызывает методы интерфейса, не будет иметь проблем, потому что останется в блаженном неведении о том, что были добавлены новые члены. Но любой класс, который реализует ваш интерфейс, до C# 8.0 перестал бы работать при добавлении новых членов. Конкретному классу требуется предоставлять все члены реализуемого интерфейса, поэтому, если интерфейс получает новые члены, полностью законченные реализации оказываются неполными. Если у вас нет какого-либо способа связаться со всеми авторами типов, реализующих ваш интерфейс, и заставить их добавить недостающие элементы, возникнут проблемы, если они обновятся до новой версии.

Вы можете решить, что трудность возникнет только в том случае, если авторы кода, работающего с интерфейсом, преднамеренно обновились до версии библиотеки с обновленным интерфейсом, так что в этот момент у них будет возможность исправить проблему. Тем не менее иногда библиотеки могут принудительно обновляться из кода. Если вы напишите приложение,

использующее несколько библиотек, каждая из которых была собрана на основе некой общей библиотеки, то по крайней мере одна из них в конечном итоге во время выполнения получит другую версию этой общей библиотеки, нежели версия, на основе которой она компилировалась. (Эталонным примером этого является библиотека JSON.NET для анализа JSON. Она чрезвычайно широко используется и имеет множество выпущенных версий, поэтому для одного приложения характерно использование нескольких библиотек, каждая из которых зависит от разных версий Json.NET. Во время выполнения используется только одна версия, поэтому не все ожидания могут оправдаться.) Это означает, что, даже если вы используете схемы, такие как семантическое управление версиями, в которых критические изменения всегда сопровождаются изменением основного номера версии компонента, этого может быть недостаточно, чтобы избежать неприятностей: вам может понадобиться использовать два компонента, где одному требуется версия 1.0 какого-то интерфейса, а другому — версия 2.0.

Результатом стало то, что интерфейсы по сути замораживались: вы не могли со временем добавлять новые члены, даже при значительных изменениях версии. Но реализации интерфейсов по умолчанию ослабляют это ограничение: вы можете добавить новый член в существующий интерфейс, если вы предоставите для него реализацию по умолчанию. Таким образом, существующие типы, которые реализовывали более старую версию, продолжали предоставлять полную реализацию обновленного интерфейса, поскольку для вновь добавленного члена в них автоматически выбиралась реализация по умолчанию без необходимости каким-либо образом ее изменять. (В этой бочке меда есть небольшая ложка дегтя, из-за чего иногда все же предпочтительнее использовать проверенное временем решение этой проблемы, т. е. абстрактные базовые классы: в главе 6 описываются эти вопросы. Таким образом, хотя реализация интерфейса по умолчанию может обеспечить полезный аварийный выход, вам все равно следует по возможности избегать изменений опубликованных интерфейсов.)

Помимо обеспечения дополнительной гибкости в вопросе обратной совместимости реализация интерфейса по умолчанию добавляет еще три возможности: теперь интерфейсы могут определять константы, статические поля и типы. Листинг 3.97 показывает интерфейс, который содержит вложенную константу и тип⁷.

⁷ В C# 8.0 вы можете вкладывать в интерфейс типы `class`, `struct`, `interface` и `enum`. Вложенные типы делегатов не поддерживаются.

Листинг 3.97. Интерфейс с константой и вложенным типом

```
public interface IContainMultitudes
{
    public const string TheMagicWord = "Please";

    public enum Outcome
    {
        Yes,
        No
    }

    Outcome MayI(string request)
    {
        return request == TheMagicWord ? Outcome.Yes : Outcome.No;
    }
}
```

При использовании подобных не похожих на метод элементов необходимо указывать видимость, поскольку в некоторых случаях вы можете добавлять эти вложенные элементы исключительно в интересах реализации методов по умолчанию, и в таком случае вы захотите, чтобы они были `private`. В данном случае я желаю, чтобы соответствующие члены были доступны для всех, так как они являются частью API, определенного интерфейсом, поэтому я пометил их как `public`. Возможно, вы смотрите на этот вложенный тип `Outcome` и задаетесь вопросом, а что, собственно, происходит? Пора положить этому конец.

Перечисления

Ключевое слово `enum` объявляет крайне простой тип, который определяет набор именованных значений. Листинг 3.98 демонстрирует перечисление, которое определяет набор взаимоисключающих выборов. Можно сказать, что оно перечисляет (*enumerates*) варианты, откуда ключевое слово `enum` и получило свое имя.

Листинг 3.98. Перечисление со взаимоисключающими вариантами

```
public enum PorridgeTemperature
{
    TooHot,
    TooCold,
    JustRight
}
```

Тип `enum` может использоваться в большинстве мест, где подойдет любой другой тип, — скажем, оно может быть типом локальной переменной, поля или параметра метода. Но один из самых распространенных способов использования `enum` — это оператор `switch`, что показано в листинге 3.99.

Листинг 3.99. Switch с перечислением

```
switch (porridge.Temperature)
{
    case PorridgeTemperature.TooHot:
        GoOutsideForABit();
        break;

    case PorridgeTemperature.TooCold:
        MicrowaveMyBreakfast();
        break;

    case PorridgeTemperature.JustRight:
        NomNomNom();
        break;
}
```

Как видно, чтобы ссылаться на члены перечисления, необходимо квалифицировать их именем типа. На самом деле `enum` — это просто причудливый способ определения загрузки полей типа `const`. Подспудно его члены представляют собой просто значения `int`. Вы можете даже явно указать их значения, как показано в листинге 3.100.

Листинг 3.100. Явные значения перечисления

```
[System.Flags]
public enum Ingredients
{
    Eggs          =      0b1,
    Bacon         =      0b10,
    Sausages      =      0b100,
    Mushrooms    =      0b1000,
    Tomato        =      0b1_0000,
    BlackPudding  =      0b10_0000,
    BakedBeans    =      0b100_0000,
    TheFullEnglish = 0b111_1111
}
```

Кроме того, в этом примере показан альтернативный способ использования `enum`. Варианты в листинге 3.100 не являются взаимоисключающими. Я ис-

пользовал здесь двоичные константы, поэтому вы можете видеть, что каждое значение соответствует определенной позиции бита, установленной в 1. Это позволяет легко объединять их — `Eggs` и `Bacon` дадут 3 (11 в двоичном виде), в то время как для `Eggs`, `Bacon`, `Sausages`, `BlackPudding` и `BakedBeans` (моя любимая комбинация) дадут 103 (1100111 в двоичном или 0x67 в шестнадцатеричном формате).

Когда вы объявляете перечисление, предназначенное для такого объединения, вы должны аннотировать его атрибутом `Flags`, определенным в пространстве имен `System`. (В главе 14 подробно описаны атрибуты.) Листинг 3.100 делает это, хотя на практике ничего особенного не случится, даже если вы забудете так поступить, потому что компилятор C# это не волнует, да и вообще, очень мало инструментов, которые обратят на это внимание. Основным преимуществом является то, что если вы вызываете `ToString` для значения `enum`, он заметит присутствие атрибута `Flags`. Для нашего типа `Ingredients` `ToString` преобразует значение 3 в строку `Eggs, Bacon`, что похоже на то, как выводит сообщения отладчик, тогда как без атрибута `Flags` оно будет обрабатываться как нераспознанное значение и вы получите строку, попросту содержащую цифру 3.



При объединении значений перечисления на основе флагов мы обычно используем побитовый оператор ИЛИ. Например, вы можете написать `Ingredients.Eggs | Ingredients.Bacon`. Это не только значительно проще читается, но также и хорошо работает с инструментами поиска Visual Studio — вы можете найти все места, где используется конкретный символ, щелкнув правой кнопкой мыши по его определению и выбрав `Find All References` из контекстного меню. Вы можете встретить код, который использует `+` вместо `|`. Это сработает для некоторых комбинаций, но `Ingredients.TheFullEnglish + Ingredients.Eggs` будет иметь значение 128, которое ничему не соответствует, поэтому безопаснее пользоваться `|`.

С таким типом перечисления в стиле битового поля у вас могут довольно быстро кончиться биты. По умолчанию для представления значения `enum` использует `int`, и в случае последовательности взаимоисключающих значений этого обычно достаточно. Сценарий, при котором потребовались бы миллиарды различных значений в одном типе перечисления, представить себе довольно сложно. Однако с 1 битом на флаг `int` позволяет задействовать только 32 флага. К счастью, можно отвоевать себе еще немногого про-

странства, указав другой базовый тип — вы можете использовать любой встроенный целочисленный тип, т. е. использовать до 64 бит. Как показано в листинге 3.101, базовый тип можно указать после двоеточия после имени типа enum.

Листинг 3.101. 64-битное перечисление

```
[System.Flags]  
public enum TooManyChoices : long  
{  
    ...  
}
```

Кстати, подобно встроенным числовым типам и любым структурам, все типы перечислений являются значимыми типами. Но они очень ограничены. Вы не можете определять какие-либо члены, кроме констант — например, методы или свойства.

Типы-перечисления способны иногда улучшать восприятие кода. Многие API-интерфейсы принимают bool для управления некоторыми сторонами своего поведения, но часто лучшим выходом был бы enum. Рассмотрим код в листинге 3.102. Он создает StreamReader, класс для работы с потоками данных, которые содержат текст. Второй аргумент конструктора — это bool.

Листинг 3.102. Бессмысленное использование bool

```
using var rdr = new StreamReader(stream, true);
```

Далеко не очевидно, что делает второй аргумент. Если вы знакомы с StreamReader, то знаете, что он определяет, должен ли порядок следования байтов в многобайтовой текстовой кодировке устанавливаться явно из кода или определяется из преамбулы в начале потока. (Здесь поможет использование синтаксиса с именованным аргументом.) И если у вас действительно хорошая память, вы даже можете вспомнить, какому из этих вариантов соответствует true. Но большинству простых смертных разработчиков, вероятно, придется обратиться к IntelliSense или даже к документации, чтобы выяснить, что же делает этот аргумент. Сравните это с листингом 3.103, который показывает другой способ.

Листинг 3.103. Ясность в перечислении

```
using var fs = new FileStream(path, FileMode.Append);
```

Второй аргумент этого конструктора использует тип перечисления, что делает код более прозрачным. Не требуется хорошей памяти, чтобы понять, что этот код намеревается добавить данные в существующий файл.

Так случилось, что этот конкретный API имеет более двух вариантов и не должен использовать `bool`. Таким образом, параметр `FileMode` действительно должен быть перечислением. Но эти примеры показывают, что даже в тех случаях, когда вы выбираете один из двух вариантов, стоит подумать об использовании перечисления, чтобы при взгляде на код было совершенно очевидно, какой именно выбор делается.

Другие типы

Мы почти закончили с нашим обзором типов и того, что в них происходит. Есть одна разновидность типа, которую я не буду затрагивать до главы 9, где речь пойдет о делегатах. Мы используем делегаты, когда нам нужна ссылка на функцию, но здесь большую роль играют детали.

Я также не упомянул указатели. C# поддерживает указатели, которые работают очень похоже на указатели в стиле С, дополненные адресной арифметикой. (Если вы не знакомы с ними, они предоставляют ссылку на определенное место в памяти.) Это немного странно, потому что они немного выбиваются из остальной системы типов. Например, в главе 2 я упомянул, что переменная типа `object` может ссылаться на «почти все». Причина, по которой я должен был это уточнить, состоит в том, что указатели являются одним из двух исключений — объект может работать с любым типом данных C#, кроме указателя или `ref struct`. (В главе 18 обсуждается последнее.)

Вот теперь мы действительно закончили. Некоторые типы в C# являются особенными, включая базовые типы, обсуждаемые в главе 2, и только что описанные структуры, интерфейсы, перечисления, делегаты и указатели, но все остальное выглядит как класс. Есть несколько классов, которые в определенных обстоятельствах обрабатываются особым образом, в частности классы атрибутов (глава 14) и классы исключений (глава 8), но за вычетом определенных особых сценариев, даже они в целом остаются самыми обычными классами. Несмотря на то что мы рассмотрели все виды типов, которые поддерживает C#, есть один способ определить класс, который я еще не показывал.

Анонимные типы

C# предлагает два механизма для группировки нескольких значений. Вы уже встречались с кортежами в главе 2. Они были введены в C# 7.0, но со временем C# 3.0 существует альтернатива: листинг 3.104 показывает, как создать и использовать экземпляр *анонимного типа*.

Листинг 3.104. Анонимный тип

```
var x = new { Title = "Lord", Surname = "Voldemort" };
Console.WriteLine($"Welcome, {x.Title} {x.Surname}");
```

Как видите, ключевое слово `new` использовано без указания имени типа. Вместо этого мы просто используем синтаксис инициализатора объекта. Компилятор C# предоставит тип, который имеет одно свойство только для чтения для каждой записи внутри инициализатора. Таким образом, в листинге 3.104 переменная `x` будет ссылаться на объект, который имеет два свойства, `Title` и `Surname`, оба типа `string`. (В анонимном типе типы свойств явно не указываются. Компилятор делает вывод о типе каждого свойства из выражения инициализации таким же образом, как и для ключевого слова `var`.) Поскольку это обычные свойства, мы можем получить к ним доступ с помощью обычного синтаксиса, как показывает последняя строка примера.

Для каждого анонимного типа компилятор генерирует довольно непримечательное определение класса. Он неизменен, потому что все свойства доступны только для чтения. Он переопределяет `Equals`, так что вы можете сравнивать экземпляры по значению, а также предоставляет соответствующую реализацию `GetHashCode`. Единственное, что необычно в сгенерированном классе, — это то, что в C# нельзя ссылаться на тип по имени. Запустив листинг 3.104 в отладчике, я обнаружил, что компилятор выбрал имя `<> f__AnonymousType0'2`. Это недопустимый идентификатор в C# из-за угловых скобок (`<>`) в начале. C# использует подобные имена всякий раз, когда нужно создать что-то, что гарантированно не будет конфликтовать с идентификаторами, которые вы можете использовать в своем собственном коде, а также когда хочет помешать вам использовать эти имена напрямую. Этот вид идентификатора довольно метко называется *непроизносимым (unspeakable) именем*.

Поскольку написать имя анонимного типа нельзя, метод не может объявить, что он его возвращает или требует в качестве аргумента (если вы не исполь-

зуете анонимный тип в качестве логического аргумента обобщенного типа, что мы увидим в главе 4). Конечно, выражение типа `object` может ссылаться на экземпляр анонимного типа, но только метод, определяющий тип, может использовать его свойства (если вы не используете тип `dynamic`, описанный в главе 2). Таким образом, анонимные типы имеют несколько ограниченное применение. Они были добавлены в язык ради LINQ, так как позволяют запросу выбирать определенные столбцы или свойства из некоторой исходной коллекции, а также определять пользовательские критерии группировки, как вы еще увидите в главе 10.

Эти ограничения дают представление о том, почему Microsoft почувствовала необходимость добавить в C# 7.0 кортежи, хотя язык уже имел довольно похожую функцию. Однако, если бы невозможность использовать анонимные типы в качестве параметров или возвращаемых типов была единственной проблемой, очевидным решением могло бы стать введение синтаксиса, позволяющего их идентифицировать. Мог бы сработать синтаксис для ссылки на кортежи — теперь мы могли бы написать (`string Name, double Age`) для ссылки на тип кортежа, так зачем же было вводить совершенно новую концепцию? Почему бы просто не использовать этот синтаксис для именования анонимных типов? (Очевидно, что мы больше не смогли бы называть их анонимными типами, но, по крайней мере, у нас не было бы двух странно схожих возможностей языка.) Однако отсутствие имен — не единственная трудность в работе с анонимными типами.

Поскольку C# используется во все более разнообразных приложениях и в более широком диапазоне аппаратных средств, производительность становится все более насущной проблемой. В сценариях доступа к базе данных, для которых анонимные типы и предназначались изначально, цена размещения объектов играла не самую заметную роль в общей картине. Однако базовая концепция — небольшой набор значений — потенциально полезна в гораздо более широком диапазоне сценариев, некоторые из которых более критичны к производительности. Однако все анонимные типы являются ссылочными, и хотя во многих случаях это не представляет трудности, некоторые сверхчувствительные к производительности сценарии могут исключать их использование. С другой стороны, кортежи являются значимыми типами, что делает их конкурентоспособными даже в коде, где вы пытаетесь минимизировать количество выделений памяти. (См. главу 7 для более подробной информации об управлении памятью и сборке мусора и главу 18 для получения информации о некоторых новых функциях языка,

направленных на более эффективное использование памяти.) Кроме того, поскольку все кортежи под капотом основаны на наборе обобщенных типов, они могут в конечном итоге уменьшить накладные расходы времени выполнения, необходимые для контроля за загруженными типами, ведь с анонимными типами вы можете получить гораздо больше загруженных типов. По связанным причинам анонимные типы будут иметь проблемы с совместимостью за границами компонента.

Значит ли это, что анонимные типы больше не нужны? На самом деле у них еще остаются кое-какие преимущества. Наиболее важным является то, что вы не можете использовать кортеж в лямбда-выражении, которое будет преобразовано в дерево выражений. Эта проблема подробно описана в главе 9, но практический итог состоит в том, что вы не можете использовать кортежи в тех запросах LINQ, о которых упоминалось ранее и для поддержки которых и были добавлены анонимные типы.

Менее очевидным является тот факт, что с кортежами имена свойств являются лишь удобной фикцией, тогда как в случае анонимных типов они вполне реальны. Из этого следует два вывода. Один касается эквивалентности: кортежи (`X: 10, Y: 20`) и (`W: 10, H: 20`) считаются взаимозаменяемыми, когда любая переменная, способная содержать один, способна содержать и другой. Это не так для анонимных типов: `new {X = 10, Y = 20}` имеет тип, отличный от `new {W = 10, H = 20}`, и попытка передать один из них в код, который ожидает другой, вызовет ошибку компилятора. Это различие может сделать кортежи более удобными в использовании, но может также сделать их более подверженными ошибкам, потому что компилятор смотрит только на форму данных, когда определяет, используете ли вы правильный тип. Анонимные типы решают эту проблему: если у вас есть два типа с одинаковыми именами и типами свойств, но семантически различающиеся, то нет способа выразить это с помощью анонимных типов. (На практике вы, вероятно, просто определите два обычных типа.) Второй итог введения анонимных типов, содержащих подлинные свойства, заключается в том, что вы можете передать их в код, который проверяет свойства объекта. Многие управляемые отражением функции, такие как определенные платформы сериализации или привязка данных инфраструктуры пользователяского интерфейса, зависят от возможности обнаружения свойств во время выполнения посредством отражения (см. главу 13). Анонимные типы могут работать с этими структурами лучше кортежей, в которых истинными именами свойств являются `Item1`, `Item2` и т. д.

Частичные типы и методы

Осталась последняя тема, которую я хочу обсудить. C# поддерживает то, что называется *частичным объявлением типа* (partial type declaration). Это очень простая концепция: объявление типа может охватывать несколько файлов. Если вы добавите ключевое слово `partial` в объявление типа, C# не будет жаловаться, если другой файл определяет тот же тип — он просто будет действовать так, как если бы все члены, определенные этими двумя файлами, появились в одном объявлении в одном файле.

Этот функционал существенно облегчает написание инструментов генерации кода. Различные функции в Visual Studio могут за вас генерировать фрагменты вашего класса. Это особенно характерно для пользовательских интерфейсов. Приложения с пользовательским интерфейсом обычно имеют разметку, которая определяет макет и содержимое каждой части пользовательского интерфейса, а вы можете выбрать, какие элементы пользовательского интерфейса будут доступны в вашем коде. Обычно это достигается добавлением поля в класс, связанный с файлом разметки. Для простоты все части этого класса, которые генерирует Visual Studio, помещаются в отдельный файл, отличный от того, куда вы пишете свою часть. Это означает, что сгенерированные части могут быть в случае необходимости переделаны с нуля, без риска перезаписи вашего кода. До того как частичные типы появились в C#, весь код для класса должен был находиться в одном файле и время от времени инструменты генерации кода путались, что приводило к потере кода.



Частичные классы не ограничиваются сценариями генерации кода, поэтому вы, конечно, можете использовать этот механизм для распределения ваших собственных определений классов по нескольким файлам. Но если вы написали такой большой и сложный класс, что чувствуете необходимость разделить его на несколько исходных файлов лишь для управляемости, это явный признак того, что класс слишком сложен. Лучшим вариантом решения этой проблемы может стать изменение дизайна классов. Тем не менее данный функционал может оказаться полезным, если вам нужно поддерживать код, собираемый по-разному для разных целевых платформ: вы можете использовать частичные классы, чтобы поместить специфичные для целевых платформ части в отдельные файлы.

Частичные методы также предназначены для сценариев генерации кода, но они немного сложнее. Они позволяют одному файлу, обычно сгенерированному, объявлять метод, а другому файлу — реализовывать его. (Строго говоря, объявление и реализация могут находиться в одном и том же файле, но обычно так не бывает.) Это может звучать как связь между интерфейсом и классом, реализующим этот интерфейс, но это не совсем то же самое. При использовании частичных методов объявление и реализация находятся в одном классе, а в разных файлах они только потому, что сам класс разделен на несколько файлов.

Если вы не предоставляете реализацию частичного метода, компилятор действует так, как будто метода вообще нет, и любой код, который вызывает метод, просто игнорируется во время компиляции. Основная причина этого заключается в поддержке механизмов генерации кода, которые могут выдавать множество видов уведомлений, но при этом вы хотите, чтобы для уведомлений, которые вам не нужны, не требовалось никаких дополнительных ресурсов. Частичные методы дают такую возможность, позволяя генератору кода объявлять частичный метод для каждого вида уведомлений, который он предоставляет, и генерировать код, который вызывает все эти частичные методы там, где это необходимо. Весь код, относящийся к уведомлениям, для которых вы не пишете метод-обработчик, будет удален во время компиляции.

Это уникальный механизм, но он направлен на платформы, которые представляют чрезвычайно детализированные уведомления и точки расширения. Вместо этого вы можете использовать некоторые более очевидные методы среди выполнения, такие как интерфейсы или функции, о которых я расскажу в следующих главах, а также обратные вызовы или виртуальные методы. Но любой из них повлечет за собой относительно высокие затраты на неиспользуемые функции. Неиспользуемые частичные методы удаляются во время компиляции, сводя к минимуму затраты на части, которые вы не используете. А это уже значительный шаг вперед.

Итог

Теперь вы имеете представление о большинстве типов, которые можете использовать в C#, а также типах членов, которые они поддерживают. Классы используются наиболее широко, но структуры полезны, когда нужна похожая

на значения семантика для присваивания и аргументов; оба поддерживают одни и те же типы членов, а именно поля, конструкторы, методы, свойства, индексаторы, события, пользовательские операторы и вложенные типы. Интерфейсы являются абстрактными, поэтому на уровне экземпляра поддерживают только методы, свойства, индексаторы и события, но с новой функцией реализации интерфейса по умолчанию в C# 8.0 они теперь способны содержать статические поля, вложенные типы и реализации по умолчанию для других членов. Перечисления же очень ограничены и предоставляют только набор известных значений.

Есть еще одна особенность системы типов C#, которая позволяет писать очень гибкие типы, называющиеся обобщенными типами. Мы рассмотрим их в следующей главе.

ГЛАВА 4

Обобщения

В главе 3 я показал, как писать типы, а также описал различные виды членов, которые они могут содержать. Однако у классов, структур, интерфейсов и методов есть еще одно измерение, о котором я пока ничего не сказал. Они позволяют определять *параметры типа*, заполнители, которые позволяют подключать различные типы во время компиляции. Это позволяет писать только один тип, а затем создавать несколько его версий. Такой тип называется *обобщенным*. Например, библиотека классов определяет обобщенный класс `List<T>`, который действует как массив переменной длины. Здесь `T` является параметром типа, и вы можете использовать почти любой тип в качестве аргумента, поэтому `List<int>` — это список целых чисел, `List<string>` — список строк и т. д. Кроме того, можно написать обобщенный метод, который имеет свои собственные аргументы типа независимо от того, является ли содержащий его тип обобщенным.

Обобщенные типы и методы визуально различимы, потому что после имени они всегда имеют угловые скобки (`<` и `>`). В них содержится разделенный запятыми список параметров или аргументов. Здесь применимо то же различие параметров/аргументов, что и в случае методов: в объявлении указывается список параметров, а затем, когда вы используете метод или тип, вы указываете уже аргументы для этих параметров. Следовательно, `List<T>` определяет один тип параметра (`T`), а `List<int>` предоставляет для этого параметра аргумент типа (`int`).

Параметры типа можно вызывать как угодно, в пределах обычных ограничений для идентификаторов в C#. Существует общее, но не универсальное соглашение об использовании `T`, когда есть только один параметр. В обобщениях с несколькими параметрами вы, скорее всего, увидите несколько более описательные имена. Например, библиотека классов определяет класс коллекции `Dictionary< TKey, TValue >`.

Иногда вы увидите подобное описательное имя, даже когда имеется лишь один параметр. И в любом случае вы, скорее всего, увидите префикс `T`, слу-

жащий для того, чтобы параметры типа выделялись, когда вы используете их в своем коде.

Обобщенные типы

Как и делегаты, которые мы рассмотрим в главе 9, классы, структуры и интерфейсы могут быть обобщенными. Листинг 4.1 показывает, как определить обобщенный класс. Синтаксис для структур и интерфейсов практически одинаков: сразу после имени типа следует список параметров типа.

Листинг 4.1. Определение обобщенного класса

```
public class NamedContainer<T>
{
    public NamedContainer(T item, string name)
    {
        Item = item;
        Name = name;
    }

    public T Item { get; }
    public string Name { get; }
}
```

Параметр типа `T` внутри тела класса можно использовать везде, где вы обычно встречаете имя типа. В этом случае я использовал его как тип аргумента конструктора, а также как свойство `Item`. Я мог бы определить и поля типа `T`. (На самом деле я это сделал, хотя и неявно. Синтаксис автоматического свойства генерирует скрытые поля, поэтому мое свойство `Item` будет иметь ассоциированное скрытое поле типа `T`.) Вы также можете определить локальные переменные типа `T`. И вы можете использовать параметры типа в качестве аргументов для других обобщенных типов. Например, мой `NamedContainer<T>` может объявить переменную типа `List<T>`.

Класс, который определяется в листинге 4.1, как и любой обобщенный тип, не является полным. Объявление обобщенного типа является несвязанным, т. е. в нем имеются параметры типа, которые необходимо заполнить, чтобы получить полный тип. Базовые вопросы, например о количестве памяти для экземпляра `NamedContainer<T>`, останутся без ответа, если не знать, что же такое `T`. Скрытым полю для свойства `Item` потребуется 4 байта, если `T` — это `int`, но 16 байт, если `decimal`. CLR не может создать исполняемый код для типа, если ему не известно, как содержимое будет организовано в памяти.

Поэтому, чтобы использовать этот или любой другой обобщенный тип, мы должны предоставить аргументы типа. Листинг 4.2 показывает, как это делается. Когда предоставляются аргументы типа, результат иногда называют *сконструируемым типом*. (Это не имеет никакого отношения к конструкторам, о которых мы говорили в главе 3. Фактически в листинге 4.2 они используются — код вызывает конструкторы двух сгенерированных типов.)

Листинг 4.2. Использование обобщенного класса

```
var a = new NamedContainer<int>(42, "The answer");
var b = new NamedContainer<int>(99, "Number of red balloons");
var c = new NamedContainer<string>("Programming C#", "Book title");
```

Вы можете использовать конструируемый обобщенный тип везде, где используете обычный. Например, вы можете использовать его в качестве типа для параметров метода и возвращаемых значений, свойств или полей. Вы можете даже использовать один в качестве аргумента типа для другого обобщенного типа, как показано в листинге 4.3.

Листинг 4.3. Конструируемый обобщенный тип в качестве аргумента типа

```
// ... где a и b взяты из листинга 4.2.
var namedInts = new List<NamedContainer<int>>() { a, b };
var namedNamedItem = new NamedContainer<NamedContainer<int>>(a, "Wrapped");
```

Каждый отдельный тип, который я предоставляю в качестве аргумента для `NamedContainer<T>`, создает отдельный тип. (А для обобщенных типов с аргументами нескольких типов каждая отдельная комбинация аргументов типа создаст отдельный тип.) Это означает, что `NamedContainer<int>` — это другой тип, нежели `NamedContainer<string>`. Вот почему можно использовать `NamedContainer<int>` как аргумент типа для другого `NamedContainer`, как это сделано в последней строке листинга 4.3, не получая при этом бесконечной рекурсии.

Поскольку каждый различный набор аргументов типа создает отдельный тип, в большинстве случаев нет подразумеваемой совместимости между различными формами одного и того же обобщенного типа. Вы не можете назначить `NamedContainer<int>` в переменную типа `NamedContainer<string>` или наоборот. В такой несовместимости есть смысл, потому что `int` и `string` — это совершенно разные типы. Но что, если использовать в качестве аргумента типа `object`? Как описано в главе 2, в переменную типа `object` вы можете поместить почти все. Если вы пишете метод с параметром типа `object`,

вполне допустимо передать туда `string`, поэтому вы можете ожидать, что метод, который принимает `NamedContainer<object>`, с радостью примет и `NamedContainer<string>`. Это не сработает, но некоторые обобщенные типы (в частности, интерфейсы и делегаты) могут объявить, что им требуется такая совместимость. Механизмы, которые поддерживают это (так называемые ковариантность и контравариантность), тесно связаны с механизмами наследования системы типов. Глава 6 посвящена наследованию и совместимости типов, поэтому там я расскажу об этом аспекте обобщенных типов.

Количество параметров типа является частью идентификатора несвязанного обобщенного типа. Это позволяет вводить несколько типов с одинаковыми именами, если они имеют разное количество параметров типа. (Технический термин для числа параметров типа — арность.) Таким образом, вы можете, скажем, определить обобщенный класс `Operation<T>`, а затем `Operation<T1, T2>`, `Operation<T1, T2, T3>`. Все это можно делать в одном пространстве имен, и это не приведет ни к какой двусмысленности. Когда вы используете эти типы, из количества аргументов ясно, какой тип имелся в виду — `Operation<int>` использует первый, в то время как `Operation<string, double>` — второй. И по той же причине необобщенный класс `Operation` будет отличаться от обобщенных типов с тем же именем.

Мой пример `NamedContainer<T>` ничего не делает с экземплярами аргумента своего типа `T` — он никогда не вызывает никаких методов, не использует каких-либо свойств или других членов `T`. Все, что он делает, так это принимает `T` в качестве аргумента конструктора, который сохраняет для последующего извлечения. Это также верно для многих обобщенных типов в библиотеке классов .NET — я упомянул некоторые классы коллекций, которые все являются вариациями одной и той же идеи о содержании данных для последующего извлечения. Для этого есть причина: обобщенный класс может работать с любым типом, поэтому он мало что может предположить касательно своих аргументов типа. Тем не менее это не жесткое правило. Вы можете задать ограничения для своих аргументов типа.

Ограничения

C# позволяет вам указать, что аргумент типа должен удовлетворять определенным требованиям. Например, предположим, что вы хотите иметь возможность создавать новые экземпляры типа по требованию. В листинге 4.4 показан простой класс, который реализует отложенное конструирование —

он делает экземпляр доступным через статическое свойство, но не пытается создать его до тех пор, пока вы не обратитесь к нему в первый раз.

Листинг 4.4. Создание нового экземпляра параметризованного типа

```
// Только для иллюстрации. Подумайте о том, как использовать Lazy<T>
// в реальной программе.
public static class Deferred<T>
    where T : new()
{
    private static T _instance;

    public static T Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new T();
            }
            return _instance;
        }
    }
}
```



На практике вы бы не стали писать подобный класс, потому что библиотека классов предлагает `Lazy<T>`, который делает ту же работу, но с предоставлением большей гибкости. `Lazy<T>` способен корректно работать в многопоточном коде, что листинг 4.4 делать не умеет. Листинг 4.4 просто иллюстрирует, как работают ограничения. Не используйте его!

Чтобы быть полезным, он должен иметь возможность создавать экземпляр любого типа, который предоставляется в качестве аргумента для `T`. Метод доступа `get` использует ключевое слово `new`, и, поскольку он не передает аргументов, он явно требует, чтобы `T` предоставил конструктор без параметров. Но не все типы это делают. Так что же произойдет, если мы попытаемся использовать тип без подходящего конструктора в качестве аргумента для `Deferred<T>`? Компилятор отклонит его, потому что он нарушает ограничение, которое обобщенный тип объявил для `T`. Ограничения располагаются непосредственно перед открывающей скобкой класса и начинаются с ключевого слова `where`. Ограничение `new ()` в листинге 4.4 утверждает, что `T` должен предоставить конструктор с нулевым аргументом.

Если бы этого ограничения не было, класс в листинге 4.4 не скомпилировался бы — вас ждала бы ошибка в строке, которая пытается создать новый `T`. Обобщенному типу (или методу) разрешено использовать только функции параметров своего типа, которые он указал посредством ограничений или которые определены базовым типом объекта. (Тип объекта определяет, например, метод `ToString`, так что вы можете вызывать его для экземпляров любого типа без необходимости указания ограничения.)

C# предлагает очень ограниченный выбор ограничений. Например, нельзя затребовать конструктор, который принимает аргументы. Фактически C# поддерживает только шесть видов ограничений для аргумента типа: ограничение типа, ограничение ссылочного типа, ограничение значимого типа, ограничение типа, не допускающего значение `NULL`, неуправляемое ограничение и ограничение конструктора без параметров. Мы уже успели столкнуться с последним, так что давайте взглянем и на остальные.

Ограничения типа

Можно так ограничить аргумент для параметра типа, чтобы он был совместим с конкретным типом. Например, вы можете потребовать, чтобы тип аргумента реализовывал определенный интерфейс. Листинг 4.5 показывает синтаксис.

Листинг 4.5. Использование ограничения типа

```
using System;
using System.Collections.Generic;

public class GenericComparer<T> : IComparer<T>
    where T : IComparable<T>
{
    public int Compare(T x, T y)
    {
        return x.CompareTo(y);
    }
}
```

Прежде чем описывать, как пример использует ограничение типа, я объясню его задачу. Этот класс обеспечивает мост между двумя стилями сравнения значений, которые вы найдете в .NET. Некоторые типы данных предоставляют собственную логику сравнения, но иногда полезно, чтобы сравнение было отдельной функцией, реализованной в своем собственном классе. Эти два стиля представлены интерфейсами `IComparable<T>` и `IComparer<T>`,

которые являются частью библиотеки классов. (Они находятся в пространствах имен `System` и `System.Collections.Generics` соответственно.) Я показал `IComparer<T>` в главе 3 – реализация этого интерфейса умеет сравнивать два объекта или значения типа `T`. Интерфейс определяет единственный метод `Compare`, который принимает два аргумента и возвращает отрицательное число, 0 или положительное число, если первый аргумент соответственно меньше, равен или больше второго. `IComparable<T>` на него очень похож, но его метод `CompareTo` принимает лишь один аргумент, потому что с помощью этого интерфейса вы просите один экземпляр сравнить себя с каким-либо другим экземпляром.

Некоторые классы коллекций библиотеки классов .NET требуют, чтобы вы реализовали `IComparer<T>` для поддержки операций упорядочивания, таких как сортировка. Они используют модель, в которой сравнение выполняет отдельный объект, потому что это имеет два преимущества перед моделью `IComparable<T>`. Во-первых, позволяет вам использовать типы данных, которые не реализуют `IComparable<T>`. Во-вторых, позволяет включать различный порядок сортировки. (Например, предположим, что вы хотите отсортировать строки с учетом регистра. Тип `string` реализует `IComparable<string>`, но он обеспечивает регистрозависимый и зависящий от местности порядок сортировки.) Так что `IComparer<T>` – это более гибкая модель. Но что, если вы используете тип данных, который реализует `IComparable<T>`, и при этом совершенно довольны упорядочиванием, которое он обеспечивает? Что вам делать, если вы работаете с API, который требует `IComparer<T>`?

На самом деле ответ таков: вы, вероятно, просто используете функцию .NET, разработанную для подобного сценария: `Comparer <T>.Default`. Если `T` реализует `IComparable<T>`, то это свойство вернет `IComparer<T>`, который делает именно то, что вы хотите. Поэтому на практике вам не нужно писать код листинга 4.5, потому что Microsoft уже написала его за вас. Однако на практике написать свою версию было бы полезным, так как на ее основе можно научиться использовать ограничения.

Строка, начинающаяся с ключевого слова `where`, гласит, что для реализации `IComparable<T>` этому обобщенному классу требуется аргумент для его параметра типа `T`. Без этого дополнения метод `Compare` не будет компилироваться, так как он вызывает метод `CompareTo` для аргумента типа `T`.

Этот метод присутствует не во всех объектах, и компилятор C# допускает его использование только потому, что мы ограничили `T` быть реализацией интерфейса, предлагающего такой метод.

Интерфейсные ограничения несколько необычны. Если методу нужен конкретный аргумент для реализации определенного интерфейса, обычно не требуется ограничение обобщенного типа. Можно просто использовать этот интерфейс в качестве типа аргумента. Однако листинг 4.5 не может этого сделать. Вы можете увидеть это, безуспешно попробовав скомпилировать листинг 4.6.

Листинг 4.6. Не компилируется: интерфейс не реализован

```
public class GenericComparer<T> : IComparer<T>
{
    public int Compare(IComparable<T> x, T y)
    {
        return x.CompareTo(y);
    }
}
```

Компилятор пожалуется на то, что не реализован метод `Compare` интерфейса `IComparer<T>`. В листинге 4.6 есть метод `Compare`, но его определение неверное — первым аргументом должен быть `T`. Можно попробовать использовать правильную сигнатуру без указания ограничения, как показано в листинге 4.7.

Листинг 4.7. Не компилируется: отсутствует ограничение

```
public class GenericComparer<T> : IComparer<T>
{
    public int Compare(T x, T y)
    {
        return x.CompareTo(y);
    }
}
```

Этот фрагмент также не удастся скомпилировать, потому что компилятор не может найти тот метод `CompareTo`, который я пытаюсь использовать. Именно ограничение для `T` в листинге 4.5 позволяет компилятору понять, что это за метод на самом деле.

Кстати, ограничения типа не обязательно должны быть интерфейсами. Можно использовать любой тип. Например, вы можете задать такое ограничение, чтобы конкретный аргумент всегда был производным от определенного базового класса. Более того, вы также можете определить ограничение одного параметра в терминах параметра другого типа. Листинг 4.8 требует, чтобы аргумент первого типа был производным от второго.

Листинг 4.8. Ограничение: один аргумент должен быть производным другого

```
public class Foo<T1, T2>
    where T1 : T2
    ...
```

Ограничения типа довольно специфичны — они требуют либо определенных отношений наследования, либо реализации определенных интерфейсов. Однако вы можете определить чуть менее конкретные ограничения.

Ограничения ссылочного типа

Вы можете ограничить аргумент типа ссылочным типом. Как показано в листинге 4.9, внешне это похоже на ограничение типа. Вы просто помещаете ключевое слово `class` вместо имени типа. Если вы используете C# 8.0 и находитесь во включенном контексте с заметками о допустимости значения `NULL`, смысл этой аннотации меняется: она требует, чтобы аргумент типа был не допускающим `NULL` ссылочным типом. Если вы укажете `class?`, это позволит типу аргумента быть как допускающим значение `NULL`, так и не допускающим значение `NULL` ссылочным типом.

Листинг 4.9. Ограничение, требующее ссылочного типа

```
public class Bar<T>
    where T : class
    ...
```

Это ограничение предотвращает использование значимых типов, таких как `int`, `double` или любой `struct` в качестве аргумента типа. Его присутствие позволяет вашему коду делать три вещи, которые иначе были бы невозможны. Во-первых, вы можете написать код, который проверяет, равны ли переменные соответствующего типа значению `null`. Если вы не ограничивали тип ссылочным типом, всегда есть вероятность, что это значимый тип, а они не могут иметь значение `null`¹. Вторая возможность заключается в том, что вы можете использовать его в качестве целевого типа оператора `as`, который мы рассмотрим в главе 6. На самом деле это всего лишь вариант первой

¹ Это разрешено, даже если вы использовали простое ограничение класса во включенном контексте с заметками о допустимости значения `NULL`. Функционал ссылок, допускающих значение `NULL`, не дает железных гарантий того, что значением ссылки не будет `null`, поэтому позволяет сравнивать ее со значением `null`.

функции — для ключевого слова `as` требуется ссылочный тип, поскольку он может возвращать `null`.



Вы не можете использовать тип, допускающий значение `NULL`, такой как `int?` (или `Nullable<int>`, как его называет CLR), в качестве аргумента для параметра с ограничением класса. Хотя вы можете проверить `int?` на значение `null` и использование его с оператором `as`, компилятор в обоих случаях генерирует совершенно иной код для типов, допускающих значение `NULL`, нежели для ссылочного типа. При использовании этих функций он не может скомпилировать один метод, который может справиться как с ссылочными типами, так и с типами, допускающими значение `NULL`.

Третья функция, которую разрешает ограничение ссылочного типа, — это возможность использовать некоторые другие обобщенные типы. Для обобщенного кода часто удобно использовать один из его аргументов типа в качестве аргумента для другого обобщенного типа, и если этот другой тип задает ограничение, вам нужно будет установить такое же ограничение для вашего собственного параметра типа. Так что если какой-то другой тип определяет ограничение класса, это может потребовать от вас ограничить один из ваших собственных аргументов таким же образом.

Конечно, это поднимает вопрос о том, почему тип, который вы используете, вообще нуждается в ограничении. Может быть, он просто хочет проверить значение на `null` или использовать оператор `as`, но тем не менее есть еще одна причина для этого ограничения. Иногда вам просто нужно, чтобы аргумент типа был ссылочным типом, — есть ситуации, в которых обобщенный метод может скомпилироваться без ограничения класса, но он не будет работать правильно, если используется со значимым типом. Чтобы было понятнее, я опишу сценарий, в котором мне иногда нужно использовать такого рода ограничения.

Я регулярно пишу тесты, которые создают экземпляр тестируемого класса, а ему, в свою очередь, нужен один или несколько фальшивых объектов вместо реальных, с которыми тестируемый объект хочет взаимодействовать. Использование этих резервных элементов уменьшает объем кода, который должен выполнять каждый отдельный тест, и способен упростить проверку поведения тестируемого объекта. Например, мой тест может потребовать проверки того, что код отправляет сообщения на сервер в нужный момент, но

я не хочу запускать реальный сервер во время юнит-теста. По этой причине я предоставляю объект, который реализует тот же интерфейс, что и класс, который будет передавать сообщение, но который не будет на самом деле отправлять никаких сообщений. Эта комбинация тестируемого объекта и фальшивого является настолько распространенной схемой, что может оказаться полезным поместить код в повторно используемый базовый класс. Использование обобщений означает, что класс может работать для любой комбинации тестируемого и фальсифицируемого типа. Листинг 4.10 показывает упрощенную версию своего рода вспомогательного класса, который я иногда пишу в подобных ситуациях.

Листинг 4.10. Ограничение другим ограничением

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

public class TestBase<TSubject, TFake>
{
    where TSubject : new()
    where TFake : class
{
    public TSubject Subject { get; private set; }
    public Mock<TFake> Fake { get; private set; }

    [TestInitialize]
    public void Initialize()
    {
        Subject = new TSubject();
        Fake = new Mock<TFake>();
    }
}
```

Существуют различные способы создания поддельных объектов для целей тестирования. Вы можете просто написать новые классы, которые реализуют тот же интерфейс, что и ваши реальные объекты. Но есть и сторонние библиотеки, которые способны их генерировать. Одна из таких библиотек называется Moq (проект с открытым исходным кодом, доступный бесплатно по адресу <https://github.com/Moq/>), и именно оттуда взят класс `Mock<T>` в листинге 4.10. Он способен генерировать фальшивую реализацию любого интерфейса или любого незапечатанного класса. (Глава 6 описывает ключевое слово `sealed`.) По умолчанию она будет содержать пустые реализации всех членов, и при необходимости вы сможете настроить интересующие вас варианты поведения. Вы также можете проверить, использовал ли тестируемый код фальшивый объект так, как вы того ожидали.

Какое все это имеет отношение к ограничениям? Класс `Mock<T>` задает ограничение ссылочного типа в своем собственном аргументе типа `T`. Это связано с тем, как он создает динамические реализации типов во время выполнения; это техника, которая работает только для ссылочных типов. Moq генерирует тип во время выполнения, и если `T` является интерфейсом, то этот сгенерированный тип будет реализовывать его, тогда как если `T` является классом, сгенерированный тип будет его производным. Он не может сделать ничего полезного, если `T` является структурой, потому что вы не можете наследовать от значимого типа². Это означает, что при использовании `Mock <T>` в листинге 4.10 мне нужно убедиться, что передаваемый аргумент типа не является структурой (это должен быть ссылочный тип). Но аргумент типа, который я использую, является одним из параметров типа моего класса: `TFake`. Так что я не знаю, какой это будет тип — это будет зависеть от того, кто использует мой класс.

Чтобы мой класс компилировался без ошибок, я должен убедиться, что выполнил ограничения всех обобщенных типов, которые я использую. Я должен гарантировать, что `Mock<TFake>` допустим, а единственный способ сделать это — добавить ограничение для моего собственного типа, которое требует, чтобы `TFake` был ссылочным типом. Что я и сделал в третьей строке определения класса в листинге 4.10. Без этого компилятор сообщит об ошибках в двух строках, которые ссылаются на `Mock<TFake>`.

В общем, если вы хотите использовать один из ваших собственных параметров типа в качестве аргумента типа для обобщенного элемента, который задает ограничение, вам необходимо указать то же самое ограничение для вашего собственного параметра типа.

Ограничения значимого типа

Так же как вы можете ограничить аргумент типа ссылочным типом, вы можете ограничить его значимым типом. Как показано в листинге 4.11, синтаксис при этом похож на синтаксис ограничения ссылочного типа, но использует ключевое слово `struct`.

² Для создания этого типа Moq использует функционал *динамических прокси-объектов* из Castle Project. Если вы хотите использовать нечто подобное в своем коде, вы можете перейти на <http://castleproject.org/>.

Листинг 4.11. Ограничение, требующее значимый тип

```
public class Quux<T>
    where T : struct
    ...
```

До этого мы видели ключевое слово `struct` только в контексте пользовательских значимых типов, но, несмотря на то, как оно выглядит, это ограничение допускает использование любых встроенных числовых типов, таких как `int`, а также пользовательских структур.

Тип .NET `Nullable<T>` накладывает именно такое ограничение. Вспомните из главы 3, что `Nullable<T>` предоставляет обертку для типов значений, которая позволяет переменной содержать либо не содержать значение. (Обычно мы используем специальный синтаксис C#, поэтому пишем, скажем, `int?` вместо `Nullable<int>`.) Единственная причина, по которой этот тип существует, состоит в том, чтобы обеспечить возможность содержать значение `NULL` для типов, которые иначе не могли бы этого делать. Так что имеет смысл использовать это только со значимым типом, так как переменные ссылочного типа уже могут быть установлены в `null` без использования этой обертки. Ограничение значимого типа не позволяет использовать `Nullable<T>` с типами, для которых это не нужно.

Значимые типы с неуправляемыми ограничениями

Вы можете указать в качестве ограничения `unmanaged`, что потребует, чтобы аргумент типа был не только значимым типом, но и не содержал ссылок. Это не только означает, что все поля типа должны быть значимыми типами, но тип каждого поля должен, в свою очередь, содержать только поля, являющиеся значимыми типами, и т. д. до самого конца. На практике это означает, что все данные должны принадлежать к фиксированному набору встроенных типов (главным образом это все числовые типы, `bool` и указатели) либо к типу `enum`. В основном это представляет интерес для сценариев взаимодействия, поскольку типы, которые соответствуют неуправляемому ограничению, могут безопасно и эффективно передаваться в неуправляемый код.

Ограничения по содержанию значения NULL

В C# 8.0 представлен новый тип ограничения, `notnull`, который доступен, если вы используете новый функционал ссылок, допускающих значение

`NULL`. Если вы его укажете, то допустимыми будут либо значимые типы, либо ссылочные типы, не допускающие значение `NULL`.

Другие специальные ограничения типа

В главе 3 описаны различные специальные типы типов, включая типы-перечисления (`enum`) и типы-делегаты (подробно описаны в главе 9). Иногда полезно ограничить аргументы типа одним из таких типов. Никаких особых хитростей тут нет: вы можете просто использовать ограничения типов. Все типы-делегаты являются производными от `System.Delegate`, а все типы перечислений — от `System.Enum`. Как показано в листинге 4.12, вы можете просто задать ограничение типа, требующее, чтобы аргумент типа был производным от любого из них.

Листинг 4.12. Ограничения, требующие делегата и перечисления

```
public class RequireDelegate<T>
    where T : Delegate
{

public class RequireEnum<T>
    where T : Enum
{
```

Раньше это не работало. Удивительно, но в течение многих лет компилятор C# старался запретить использование этих двух типов в качестве ограничений. Лишь в C #7.3 мы наконец получили возможность задавать такие ограничения.

Множественные ограничения

Если вы хотите наложить несколько ограничений для одного аргумента типа, вы можете просто поместить их в список, как показано в листинге 4.13. Есть несколько ограничений порядка следования: если у вас есть ограничение ссылочного типа или значимого типа, ключевое слово `class` или `struct` должно стоять в списке на первом месте. Если присутствует ограничение `new ()`, оно должно быть последним.

Когда ваш тип имеет несколько параметров типа, вы пишете по одному разделу `where` для каждого параметра типа, который хотите ограничить. Фак-

тически мы уже встречались с этим — листинг 4.10 определяет ограничения для обоих своих параметров.

Листинг 4.13. Множественные ограничения

```
public class Spong<T>
    where T : IEnumerable<T>, IDisposable, new()
...
```

Нулевые значения

Есть определенные функции, которые поддерживаются всеми типами, и потому не требуют ограничений. В их число входит набор методов, определяемых базовым классом `object`, описанным в главах 3 и 6. Но есть еще более базовый функционал, который иногда может быть полезен в рамках обобщенного кода.

Переменные любого типа могут быть инициализированы значением по умолчанию. Как вы уже видели в предыдущих главах, в ряде ситуаций CLR делает это за нас. Например, все поля во вновь созданном объекте будут иметь известное значение, даже если мы не будем писать инициализаторы полей и не укажем значения в конструкторе. Аналогично новый массив любого типа будет иметь все свои элементы инициализированными известным значением. CLR делает это, заполняя соответствующую память нулями. Точное значение при этом зависит от типа данных. Для любого из встроенных числовых типов значение будет в буквальном смысле числом 0, но для нечисловых типов будет уже чем-то другим. Для `bool` значение по умолчанию — `false`, а для ссылочного типа — `null`.

Иногда для обобщенного кода может оказаться полезным установить переменную в это начальное нулевое значение по умолчанию. Но в большинстве случаев вы не можете использовать для этого литеральное выражение. Вы не можете присвоить значение `null` переменной, если ее тип — параметр типа, если только он не ограничен ссылочным типом. И вы не можете присвоить такой переменной литерал 0, потому что аргумент типа невозможно связать с числовым типом.

Вместо этого вы можете запросить нулевое значение для любого типа, используя ключевое слово `default`. (Это то самое ключевое слово, которое мы видели у оператора `switch` в главе 2, но там оно использовалось совершенно по-другому. C# продолжает традицию С-семейства по определению нескольки-

ких несвязанных значений для каждого ключевого слова.) Если вы напишите `default(SomeType)`, где `SomeType` — это определенный тип или параметр типа, то получите начальное значение по умолчанию для этого типа: `0`, если это числовой тип, и его эквивалент для любого другого типа. Например, выражение `default (int)` имеет значение `0`, `default (bool)` — `false`, а `default (string)` — `null`. Используйте это ключевое слово с параметром обобщенного типа, чтобы получить значение по умолчанию для соответствующего аргумента типа, как показано в листинге 4.14.

Листинг 4.14. Получение значения по умолчанию (нулевого) для аргумента типа

```
static void ShowDefault<T>()
{
    Console.WriteLine(default(T));
}
```

Внутри обобщенного типа или метода, который определяет параметр типа `T`, выражение `default(T)` создаст для `T` нулевое значение по умолчанию — независимо от того, чем является `T`, — без каких-либо ограничений. Таким образом, вы можете использовать обобщенный метод в листинге 4.14, чтобы убедиться, что значения по умолчанию для `int`, `bool` и `string` являются указанными мной значениями.

В случаях, когда компилятор способен определить, какой тип требуется, можно использовать упрощенную форму. Вместо `default(T)` вы можете написать только `default`. Это не сработает в листинге 4.14, потому что `Console.WriteLine` может принимать практически все, поэтому компилятор не может сузить диапазон до одного параметра, но сработает в листинге 4.15, потому что компилятор видит, что обобщенный тип возвращаемого значения метода — `T`, так что для него потребуется `default(T)`. Поскольку он вполне способен это понять, нам достаточно написать `default`.

Листинг 4.15. Получение нулевого значения по умолчанию для предполагаемого типа

```
static T GetDefault<T>() => default;
```

И поскольку я только что показал вам пример обобщенного метода, похоже, сейчас самое время о них поговорить.

Обобщенные методы

Помимо обобщенных типов C# также поддерживает обобщенные методы. В этом случае список параметров обобщенного типа следует за именем метода и предшествует обычному списку параметров метода. В листинге 4.16 показан метод с единственным параметром типа. Этот параметр используется в качестве возвращаемого типа, а также в качестве типа элемента для массива, передаваемого в качестве аргумента метода. Метод возвращает последний элемент в массиве, и, поскольку он является обобщенным, он будет работать с любым типом элемента массива.

Листинг 4.16. Обобщенный метод

```
public static T GetLast<T>(T[] items) => items[items.Length - 1];
```



Вы можете определить обобщенные методы внутри как обобщенных типов, так и не обобщенных типов. Если обобщенный метод является членом обобщенного типа, все параметры типа из содержащего находятся в области действия при применении внутри метода, как и параметры типа, специфичные для метода.

Как и в случае обобщенного типа, вы можете использовать обобщенный метод, указав его имя вместе с аргументами типа, как показано в листинге 4.17.

Листинг 4.17. Вызов обобщенного метода

```
int[] values = { 1, 2, 3 };
int last = GetLast<int>(values);
```

Обобщенные методы работают аналогично обобщенным типам, но с параметрами типов, область видимости которых распространяется на объявление метода и его тело. Вы можете указать ограничения почти так же, как и с обобщенными типами. Ограничения размещаются после списка параметров метода и перед его телом, как это показано в листинге 4.18.

Листинг 4.18. Обобщенный метод с ограничением

```
public static T MakeFake<T>()
    where T : class
{
    return new Mock<T>().Object;
}
```

Есть один способ отличия обобщенных методов от обобщенных типов: вам не всегда нужно явно указывать аргументы типа обобщенного метода.

Выведение типа

Компилятор C# может самостоятельно вывести аргументы типа для обобщенного метода. Я могу изменить листинг 4.17, удалив список аргументов типа из вызова метода, как показано в листинге 4.19, и это никак не изменит суть кода.

Листинг 4.19. Вывод аргумента обобщенного типа для метода

```
int[] values = { 1, 2, 3 };
int last = GetLast(values);
```

При наличии подобного вызова такого типа, если нет необобщенного метода с таким именем, компилятор начинает поиск подходящих обобщенных методов. Если метод в листинге 4.16 находится в области видимости, он будет таким кандидатом и компилятор попытается определить аргументы типа. Это довольно простой случай. Метод ожидает массив некоторого типа T , и мы передали массив с элементами типа `int`, поэтому несложно понять, что этот код следует рассматривать как вызов `GetLast<int>`.

Но все может быть гораздо более сложным. Спецификация C# содержит около шести страниц, посвященных алгоритму вывода типов, но все это преследует одну цель: позволить вам опускать аргументы типа, когда они избыточны. Кстати, выведение типа всегда выполняется во время компиляции, поэтому основано на статическом типе аргументов метода.

В случае API, которые широко используют обобщения (такие, как LINQ, глава 10), явное перечисление каждого аргумента типа может сделать код очень сложным для понимания, поэтому обычно полагаются на вывод типа. И, если вы используете анонимные типы, вывод аргументов типа приобретает особенное значение, потому что в этом случае невозможно явно указать аргументы типа.

Обобщения и кортежи

Облегченные кортежи C# имеют особый синтаксис, но с точки зрения среды выполнения в них нет ничего особенного. Все они являются просто

экземплярами набора обобщенных типов. Взгляните на листинг 4.20. Он использует `(int, int)` в качестве типа локальной переменной, указывая, что это кортеж, содержащий два значения `int`.

Листинг 4.20. Объявление переменной кортежа обычным способом

```
(int, int) p = (42, 99);
```

Теперь посмотрим на листинг 4.21. Он использует уже `ValueTuple<int, int>` из пространства имен `System`. Но это эквивалентно объявлению в листинге 4.20. Если в Visual Studio навести указатель мыши на переменную `p2`, она покажет свой тип как `(int, int)`.

Листинг 4.21. Объявление переменной кортежа с ее базовым типом

```
ValueTuple<int, int> p2 = (42, 99);
```

Одна особенность, которую добавляет специальный синтаксис C# для кортежей, — это возможность именовать элементы кортежей. Семейство `ValueTuple` называет свои элементы `Item1`, `Item2`, `Item3` и т. д., но в C# мы можем выбрать другие имена. Когда вы объявляете локальную переменную с именованными элементами кортежа, эти имена являются видимостью, поддерживаемой C#, — их вообще не существует во время исполнения. Однако, когда метод возвращает кортеж, как в листинге 4.22, все по-другому: имена должны быть видны, чтобы код, использующий этот метод, мог их использовать. Даже если этот метод находится в компоненте библиотеки, на который ссылается мой код, я хочу иметь возможность писать `Pos().X` вместо `Pos().Item1`.

Листинг 4.22. Возвращение кортежа

```
public (int X, int Y) Pos() => (10, 20);
```

Чтобы код работал, компилятор применяет атрибут с именем `TupleElementNames` к возвращаемому значению метода, а он содержит массив, перечисляющий используемые имена свойств. (В главе 14 описываются атрибуты.) На самом деле вы не сможете самостоятельно написать код, который такое проделывает: если написать метод, который возвращает `ValueTuple <int, int>`, и попытаться применить `TupleElementNamesAttribute` в качестве атрибута возврата, компилятор выдаст ошибку, предписывающую не использовать этот атрибут напрямую, а использовать синтаксис кортежа. Но этот атрибут — способ, которым компилятор сообщает имена элементов кортежа.

Помните, что в библиотеке классов .NET есть еще одно семейство типов-кортежей, `Tuple<T>`, `Tuple<T1, T2>` и т. д. Они выглядят почти идентичными семейству `ValueTuple`. Разница в том, что семейство обобщенных типов `Tuple` — это классы, а все типы `ValueTuple` — структуры. Облегченный синтаксис кортежа C# использует только семейство `ValueTuple`. Однако семейство `Tuple` существует в библиотеках классов .NET гораздо дольше, поэтому их можно часто встретить в старом коде, который должен был объединять набор значений без определения нового типа.

Внутренние обобщения

Если вы знакомы с шаблонами C++, то заметили, что обобщения C# сильно отличаются от шаблонов. Внешне они имеют некоторые сходства и могут использоваться похожими способами — обе техники подходят для реализации классов-коллекций. Но есть некоторые основанные на шаблонах методы, которые просто не будут работать в C# (листинг 4.23).

Листинг 4.23. Техника шаблонов, которая не работает в обобщениях C#

```
public static T Add<T>(T x, T y)
{
    return x + y; // Не будет компилироваться
}
```

Такое допустимо в шаблоне C++, но не в C#, и полностью исправить это с помощью ограничения невозможно. Вы можете добавить ограничение типа, требующее, чтобы `T` наследовался от некоторого типа, который определяет пользовательский оператор `+`, что позволило бы ему компилироваться. Но в таком случае он был бы довольно ограниченным и работал только для типов, производных от этого базового типа. В C++ можно написать шаблон, который будет складывать два поддерживающих сложение элемента любого типа, будь он встроенным или пользовательским. Более того, шаблоны C++ не нуждаются в ограничениях; компилятор сам определит, будет ли определенный тип работать в качестве аргумента шаблона.

Эта проблема относится не только к арифметике. Суть в том, что обобщенный код опирается на ограничения, чтобы выяснить, какие операции доступны для его параметров типа. Поэтому он может использовать только функции, представленные в качестве членов интерфейсов или общих ба-

зовых классов. Если бы арифметика в .NET основывалась на интерфейсе, можно было бы определить ограничение, которое этого требует.

Но все операторы являются статическими методами, и, хотя C# 8.0 позволил интерфейсам содержать статические члены, у отдельных типов нет возможности предоставить собственную реализацию. Механизм динамической диспетчеризации, который позволяет каждому типу предоставлять собственную реализацию интерфейса, работает только для членов экземпляра. Эта новая языковая возможность позволяет представить некий интерфейс `IArithmetic`, определяющий необходимые методы статического оператора, все из которых полагаются на элементы — экземпляры интерфейса, которые выполняют фактическую работу, но на момент написания такого механизма не существует.

Ограничения обобщений C# обусловлены спецификой работы, поэтому полезно понять их механизм. (Кстати, эти ограничения не являются специфическими для CLR компании Microsoft. Они являются неизбежным результатом того, как обобщения вписаны в дизайн CLI.)

Обобщенные методы и типы компилируются без знания о том, какие типы будут использоваться в качестве аргументов. В этом принципиальное отличие обобщенных шаблонов C# от шаблонов C++. В C++ компилятор видит все экземпляры шаблона. Но с C# вы можете создавать экземпляры обобщенных типов без доступа к какому-либо соответствующему исходному коду еще долго после того, как код был скомпилирован. В конце концов, Microsoft написала обобщенный класс `List<T>` несколько лет назад, но вы и сейчас можете написать совершенно новый класс и использовать его в качестве аргумента типа `T`. (Стоит отметить, что стандартная библиотека C++ `std::vector` существует еще дольше. Тем не менее компилятор C++ имеет доступ к исходному файлу, который определяет класс, что не так в случае с C# и `List<T>`. C# видит только скомпилированную библиотеку.)

Результатом этого является то, что компилятору C# нужно иметь достаточно информации для генерации типобезопасного кода в момент, когда он компилирует обобщенный код. Возьмите листинг 4.23. Он не может знать, что в его случае означает оператор `+`, потому что он будет различным для разных типов. Со встроенными числовыми типами этот код должен был бы компилироваться в инструкции специализированного промежуточного языка (IL) для выполнения сложения. Если бы этот код находился в проверенном контексте (т. е. с использованием ключевого слова `checked`, опи-

санного в главе 2), у нас тут же возникла бы проблема, потому что код для сложения целых чисел с проверкой переполнения использует разные коды операций IL для знаковых и беззнаковых целых чисел. Кроме того, так как метод обобщенный, мы можем вообще не иметь дела со встроенными числовыми типами. Возможно, мы будем работать с типом, который определяет пользовательский оператор +, а в этом случае компилятору потребуется сгенерировать вызов метода. (Пользовательские операторы — это просто скрытые методы.) Или, если указанный тип не поддерживает сложение, компилятор должен выдать ошибку.

В зависимости от используемых типов существует несколько возможных вариантов компиляции простого выражения сложения. Хорошо, когда типы известны компилятору, но он должен компилировать код и для обобщенных типов и методов, не зная, какие именно типы будут использоваться в качестве аргументов.

Вы могли бы сказать, что, возможно, Microsoft стоит разработать какой-то предварительный наполовину скомпилированный формат для обобщенного кода. В некотором смысле это уже сделано. Вводя обобщения, Microsoft изменила систему типов, формат файла и инструкций IL, чтобы разрешить обобщенному коду использовать заполнители, представляющие параметры типа, которые заполняются, когда тип полностью создан. Так почему бы не расширить этот механизм для обработки операторов? Почему бы не позволить компилятору генерировать ошибки в тот момент, когда вы компилируете код, пытающийся использовать обобщенный тип, вместо того чтобы упорно выдавать ошибки, когда компилируется сам обобщенный код? Выходит, что вы можете подключать новые наборы аргументов типов во время выполнения. API отражений, который мы рассмотрим в главе 13, позволяет создавать обобщенные типы. Таким образом, компилятор не обязательно будет доступен в момент, когда ошибка станет очевидной, поскольку не все версии .NET поставляются с копией компилятора C#. И что должно произойти, если обобщенный класс написан на C#, но используется совершенно другим языком, не поддерживающим перегрузку операторов? Правила какого языка должны применяться, когда нужно решить, что делать с оператором +? Должен ли это быть язык, на котором был написан обобщенный код, или язык, на котором написан аргумент типа? (Что, если есть несколько параметров типа и для каждого аргумента используется тип, написанный на другом языке?) Или, возможно, следует использовать правила того языка, который применяется

для включения аргументов типа в обобщенный тип или метод? Но как быть в случае, когда один фрагмент обобщенного кода передает свои аргументы какой-либо другой обобщенной сущности? Даже если бы вы могли выбрать наилучший подход, это предполагает, что правила, используемые для определения того, что на самом деле означает строка кода, доступны во время выполнения, но это допущение, повторюсь, противоречит тому, что соответствующие компиляторы не обязательно будут установлены на машине с запущенным кодом.

Обобщения .NET решают эту проблему, требуя полного определения значения обобщенного кода при компиляции, используя при этом правила языка, на котором был написан обобщенный код. Если обобщенный код предполагает использование методов или других элементов, они должны быть разрешены статически (т. е. сущность этих элементов должна быть точно определена во время компиляции). Важно то, что время при этом уходит на компиляцию самого обобщенного кода, а не кода, использующего обобщенный код. Эти требования объясняют, почему обобщения C# не обладают такой гибкостью, как модель замещения во время компиляции, которую использует C++. Преимущество состоит в том, что вы можете компилировать обобщения в библиотеки в двоичной форме и они могут использоваться любым языком .NET, который поддерживает обобщения, при этом их поведение будет полностью предсказуемым.

Итог

Обобщения позволяют нам писать типы и методы с аргументами типов, которые могут быть заполнены во время компиляции. Тем самым можно создавать несколько версий типов или методов, которые работают с конкретными типами. Когда обобщения были впервые представлены, самый важный сценарий их использования состоял в написании типобезопасных классов коллекций. На заре .NET обобщений еще не было, поэтому классы коллекций, доступные с версии 1.0, использовали тип общего назначения `object`.

Это означало, что вам приходилось приводить объекты к их настоящему типу каждый раз, когда вы извлекали объект из коллекции. Это также означало, что значимые типы обрабатывались в коллекциях не самым эффективным образом; как мы увидим в главе 7, ссылка на значения через объект требует генерации *упаковок* для хранения значений. Обобщения легко справляются

с этими проблемами. Они позволяют писать классы коллекций, такие как `List<T>`, которые можно использовать без приведения. Более того, поскольку CLR может создавать обобщенные типы во время выполнения, возможно генерировать код, оптимизированный для любого типа, который содержит коллекция. Таким образом, классы коллекций теперь могут обрабатывать значимые типы, такие как `int`, гораздо эффективнее, чем до появления обобщенных типов. Мы рассмотрим некоторые из этих типов-коллекций в следующей главе.

ГЛАВА 5

Коллекции

Большинству программ приходится иметь дело с множеством фрагментов данных. Возможно, ваш код выполняет цикл по транзакциям для расчета баланса счета, или отображает недавние сообщения в веб-приложении для соцсетей, или обновляет позиции персонажей в игре. В большинстве видов приложений вам, скорее всего, пригодится возможность работать с коллекциями информации.

В C# имеется простой вид коллекции, называемый *массивом*. Система типов CLR имеет внутреннюю поддержку массивов, поэтому они эффективны и экономичны, но для некоторых сценариев могут оказаться недостаточными. Но библиотека классов, основываясь на базовых функциях массивов, предоставляет более мощные и гибкие типы коллекций. Я начну с массивов, потому что именно они являются основой большинства классов коллекций.

Массивы

Массив — это объект, который содержит несколько элементов определенного типа. Каждый элемент представляет собой место в памяти, похожее на поле, но если в случае полей мы даем имя каждому месту в памяти, элементы же массива просто нумеруются. Количество элементов фиксируется на все время жизни массива, поэтому размер нужно указывать при его создании. В листинге 5.1 приведен синтаксис создания новых массивов.

Листинг 5.1. Создание массивов

```
int[] numbers = new int[10];
string[] strings = new string[numbers.Length];
```

Как и другие объекты, массив создается с помощью ключевого слова `new`, за которым следует имя типа, но вместо скобок с аргументами конструктора пишутся квадратные скобки, содержащие размер массива. Как показывает пример, выражение, определяющее размер, может быть константой, но не

обязательно — размер второго массива будет определяться путем вычисления `numbers.Length` во время выполнения. В данном случае второй массив будет иметь 10 элементов, потому что мы используем свойство `Length` первого массива. Все массивы имеют свойство только для чтения, которое возвращает общее количество элементов в массиве.

Свойство `Length` имеет тип `int`, что означает, что оно может использоваться «только» с массивами, содержащими до 2,1 миллиарда элементов. В 32-битном процессе это редко является проблемой, поскольку ограничивающим фактором для размера массива, скорее всего, станет доступное адресное пространство. В 64-битных процессах возможны массивы большего размера, поэтому в наличии также свойство `LongLength` типа `long`. Нечасто можно увидеть, как кто-то его использует, потому что среда выполнения в настоящее время не поддерживает создание массивов с более чем 2 147 483 591 (0x7FFFFFFF) элементом в любом одном измерении. Таким образом, только прямоугольные многомерные массивы (описанные далее в этой главе) могут содержать больше элементов, чем может позволить `Length`. Но даже у них в текущих версиях .NET есть верхний предел в 4 294 967 295 (0xFFFFFFFF) элементов.



Если вы используете .NET Framework (а не .NET Core), то столкнетесь с другим ограничением: один массив обычно не может занимать более 2 ГБ памяти. (Это верхний предел размера любого отдельного объекта. На практике такого предела достигают только массивы, хотя его можно достичь с помощью сверхдлинной строки.) Его можно преодолеть, если добавить в секцию `<runtime>` файла `App.config` элемент `<gcAllowVeryLargeObjects enabled="true" />`. Ограничения из предыдущего параграфа все еще действуют, но теперь они значительно менее строгие, чем ограничение в 2 ГБ.

В листинге 5.1 я нарушил свое обычное правило избегать избыточных имен типов в объявлениях переменных. Согласно блоку инициализации, переменные являются массивами типа `int` и `string` соответственно, поэтому, хотя в подобном случае я обычно использую `var`, здесь я сделал исключение, чтобы показать, как задать имя типа массива. Типы массивов сами по себе являются отдельными типами, и, если мы хотим сослаться на тип, который является одномерным массивом определенного типа, мы ставим `[]` после имени типа элементов.

Все типы массивов являются производными от общего базового класса `System.Array`. Он определяет свойства `Length` и `LongLength`, а также другие элементы, которые мы рассмотрим в свое время. Вы можете использовать массивы везде, где и другие типы. Таким образом, вы можете указать тип `string[]` как тип поля или параметра метода. Вы также можете использовать массив в качестве аргумента обобщенного типа. Например, `IEnumerable<int[]>` будет последовательность массивов целых чисел (каждый из которых может быть разного размера).

Массив всегда является ссылочным типом, независимо от типа элементов. Тем не менее выбор между ссылочным типом и значимым типом в качестве типа элементов значительно меняет поведение массива. Как обсуждалось в главе 3, когда у объекта есть поле значимого типа, само значение располагается в памяти, выделенной для объекта. То же верно для массивов — когда типом элементов является значимый тип, то значение располагается в элементе массива, но в случае ссылочного типа элементы содержат только ссылки. Каждый экземпляр ссылочного типа указывает на что-то, и, поскольку многие переменные могут в конечном итоге ссылаться на то же самое, CLR необходимо управлять временем жизни этой сущности независимо от любого другого объекта, поэтому она будет располагаться в собственном отдельном блоке памяти. Таким образом, хотя массив из 1000 значений типа `int` может находиться в одном сплошном блоке памяти, в случае ссылочных типов массив содержит только ссылки, а не фактические экземпляры. Массиву 1000 различных строк потребуется 1001 блок в куче — один для самого массива и один для каждой строки.



При использовании элементов ссылочного типа не обязательно делать так, чтобы каждый элемент в массиве ссылок ссылался на отдельный объект. Можно оставить любое количество элементов равными `null`, а также сделать так, чтобы несколько элементов ссылались на один и тот же объект. Это просто еще одно указание на то, что ссылки в элементах массива работают почти так же, как в локальных переменных и полях.

Чтобы получить доступ к элементу в массиве, мы используем квадратные скобки, содержащие индекс элемента, который нам нужен. Индексация начинается с нуля. Листинг 5.2 показывает несколько примеров.

Как и в случае с размером при создании, индекс массива может быть константой, но может быть и более сложным выражением, вычисляемым во время

выполнения. На самом деле это относится и к той части, которая находится непосредственно перед открывающей скобкой. В листинге 5.2 для ссылки на массив я использовал переменную, но вы можете использовать скобки после любого выражения, тип которого является массивом. Листинг 5.3 получает первый элемент массива, возвращаемого методом. (Подробности работы кода не имеют особого значения, но если вам интересно, то он находит сообщение об авторских правах, связанное с компонентом, который определяет тип объекта. Например, если вы передадите методу `string`, то он вернет «© Microsoft Corporation. All rights reserved.». При этом используются API отражения и пользовательские атрибуты — темы глав 13 и 14.)

Листинг 5.2. Доступ к элементам массива

```
// продолжение листинга 5.1
numbers[0] = 42;
numbers[1] = numbers.Length;
numbers[2] = numbers[0] + numbers[1];
numbers[numbers.Length - 1] = 99;
```

Листинг 5.3. Сложный доступ к массиву

```
public static string GetCopyrightForType(object o)
{
    Assembly asm = o.GetType().Assembly;
    var copyrightAttribute = (AssemblyCopyrightAttribute)
        asm.GetCustomAttributes(typeof(AssemblyCopyrightAttribute),
        true)[0];
    return copyrightAttribute.Copyright;
}
```

Выражения, используемые для доступа к элементу массива, особенные в том смысле, что C# считает их своего рода переменной. Это означает, что, как и в случае с локальными переменными и полями, вы можете использовать их в левой части оператора присваивания независимо от того, просты ли они, как выражения в листинге 5.2, или более сложные, как в листинге 5.3. Вы можете также использовать их с ключевым словом `ref` (как описано в главе 3), чтобы передать ссылку на определенный элемент в метод, сохранить его в локальной переменной `ref` или вернуть его из метода как тип `ref`.

CLR всегда сравнивает индекс с размером массива. Если вы попытаетесь использовать либо отрицательный индекс, либо индекс, который больше или равен длине массива, среда выполнения выдаст исключение `IndexOutOfRangeException`.

Несмотря на то что размер массива всегда фиксирован, его содержимое всегда можно изменить — такого понятия, как массив только для чтения, не существует. (Как мы увидим в «`ReadOnlyCollection<T>`» на с. 310, .NET содержит класс, который может создавать видимость, что массив только для чтения.) Конечно, вы можете создать массив с неизменяемым типом элементов, и это предотвратит изменение. Поэтому код из листинга 5.4, в котором используется неизменяемый значимый тип `Complex` из библиотеки .NET, не будет компилироваться.

Листинг 5.4. Массив с неизменяемыми элементами изменить нельзя!

```
var values = new Complex[10];
// Эти строки вызывают ошибки компилятора:
values[0].Real = 10;
values[0].Imaginary = 1;
```

Компилятор будет жаловаться на то, что свойства `Real` и `Imaginary` доступны только для чтения; тип `Complex` не предоставляет никакой возможности изменить свое значение. Тем не менее вы можете изменить массив: даже если вы не можете в обычном порядке изменить существующий элемент, вы всегда можете перезаписать его, указав другое значение, как показано в листинге 5.5.

Листинг 5.5. Изменение массива с неизменяемыми элементами

```
var values = new Complex[10];
values[0] = new Complex(10, 1);
```

Массивы, доступные только для чтения, в целом не очень полезны, потому что все массивы изначально содержат значение по умолчанию, которое вы не можете указать. CLR заполняет память для нового массива нулями, поэтому вы увидите `0`, `null` или `false` в зависимости от типа элементов массива. Для некоторых приложений полностью нулевое (или эквивалентное) начальное содержимое элементов массива может оказаться полезным, но в некоторых случаях вы захотите перед началом работы задать какое-то другое содержимое.

Инициализация массива

Самый простой способ инициализации массива состоит в том, чтобы присваивать значения каждому элементу по очереди. В листинге 5.6 создается строковый массив, а так как строка является ссылочным типом, создание

массива из пяти элементов не создает пяти строк. Наш массив изначально содержит пять значений `null`. (Это будет так, даже если вы включите функционал ссылок с возможностью содержания значения `NULL` в C #8.0, как описано в главе 3. К сожалению, инициализация массива — это одна из дыр, которые делают невозможным предоставление абсолютных гарантий, что значение не будет `null`.) Поэтому код продолжается тем, что каждый элемент массива заполняется ссылкой на строку.

Листинг 5.6. Трудоемкая инициализация массива

```
var workingWeekDayNames = new string[5];
workingWeekDayNames[0] = "Monday";
workingWeekDayNames[1] = "Tuesday";
workingWeekDayNames[2] = "Wednesday";
workingWeekDayNames[3] = "Thursday";
workingWeekDayNames[4] = "Friday";
```

Такой код работает, но он излишне многословен. C# поддерживает более короткий синтаксис, который показан в листинге 5.7 и позволяет делать то же самое. Компилятор сам превращает это в код, похожий на листинг 5.6.

Листинг 5.7. Синтаксис инициализатора массива

```
var workingWeekDayNames = new string[]
{ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" };
```

Можно пойти дальше. Листинг 5.8 показывает, что если вы явно указываете тип в объявлении переменной, то можете написать только список инициализаторов, не используя ключевое слово `new`. Кстати, это работает только в выражениях инициализатора; нельзя использовать этот синтаксис для создания массива в других выражениях, таких как присваивание или аргументы метода. (Более явное выражение инициализатора в листинге 5.7 работает во всех этих контекстах.)

Листинг 5.8. Более короткий синтаксис инициализатора массива

```
string[] workingWeekDayNames =
{ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" };
```

Мы можем пойти еще дальше: если все выражения в списке инициализатора массива имеют одинаковый тип, компилятор может определить тип массива, поэтому мы можем написать просто `new[]` без явного указания типа элемента. Листинг 5.9 это делает.

Листинг 5.9. Синтаксис инициализатора массива с определением типа элемента

```
var workingWeekDayNames = new[ ]
    { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" };
```

Получилось чуть длиннее, чем в листинге 5.8. Но как и в листинге 5.7, подобный стиль применим не только к инициализации переменной. Например, вы также можете использовать его, когда нужно передать массив в качестве аргумента метода. Если создаваемый вами массив будет только передан в метод и на него никогда не будут ссылаться снова, вам может быть не нужна отдельная переменная для ссылки на него. Может оказаться полезнее вписать массив непосредственно в список аргументов. Листинг 5.10 передает массив строк методу, используя эту технику.

Листинг 5.10. Массив как аргумент

```
SetHeaders(new[] { "Monday", "Tuesday", "Wednesday", "Thursday",
    "Friday" });
```

Поиск и сортировка

Вы не всегда будете знать индекс нужного элемента массива. Предположим, вы пишете приложение, которое выводит список недавно использованных файлов. Каждый раз, когда пользователь открывает файл в вашем приложении, вы помещаете этот файл в начало списка, и, чтобы избежать его многократного там появления, вам нужно определить, не находится ли файл уже в списке. Если бы пользователь для открытия файла использовал ваш список недавних файлов, вы бы заранее знали, что он находится в списке, и знали его смещение. Но что, если пользователь откроет файл другим способом? В этом случае у вас есть имя файла и вам нужно выяснить, где оно расположено в вашем списке, если оно вообще там есть.

В подобном сценарии массивы помогут найти нужный элемент. Существуют методы, которые проверяют каждый элемент по очереди, останавливаясь при первом совпадении, а также методы, которые способны работать значительно быстрее, если ваш массив хранит свои элементы по порядку. А чтобы помочь с этим, есть методы для сортировки содержимого массива в любом нужном порядке.

Статический метод `Array.IndexOf` обеспечивает наиболее простой способ поиска элемента. Вам не нужно, чтобы элементы вашего массива находились в каком-либо определенном порядке: вы просто передаете ему массив

для поиска и искомое значение, а он будет проходить по элементам, пока не найдет значение, равное тому, которое вы указали. Он возвращает индекс, по которому он нашел первый соответствующий элемент, или `-1`, если достиг конца массива, не найдя соответствия. Листинг 5.11 показывает, как можно использовать этот метод в логике обновления списка недавно открытых файлов.

Листинг 5.11. Поиск с помощью `IndexOf`

```
int recentFileListIndex = Array.IndexOf(myRecentFiles, openedFile);
if (recentFileListIndex < 0)
{
    AddNewRecentEntry(openedFile);
}
else
{
    MoveExistingRecentEntryToTop(recentFileListIndex);
}
```

Этот пример начинает поиск с начала массива, но есть и другие варианты. Метод `IndexOf` перегружен, и вы можете передать индекс, от которого нужно начать поиск, и, что необязательно, второе число, указывающее, сколько элементов нужно проверить до того, как сдаться. Еще есть метод `LastIndexOf`, который работает в обратную сторону. Если вы не укажете индекс, он начнет работу с конца массива и пойдет в обратном направлении. Как и в случае `IndexOf`, вы можете указать один или два аргумента, указывающих смещение, с которого вы хотите начать, и количество проверяемых элементов.

Эти методы хороши, если вы точно знаете, какое значение ищете, но часто вам требуется немного больше гибкости: вам может понадобиться найти первый (или последний) элемент, который соответствует определенным критериям. Предположим, у вас есть массив, содержащий значения столбцов гистограммы. Вам может понадобиться выяснить, какой из них является первым ненулевым столбцом. Поэтому вместо поиска определенного значения вам нужно найти первый элемент с любым значением, отличным от нуля. В листинге 5.12 показано, как использовать метод `FindIndex` для поиска первой такой записи.

Мой метод `IsNonZero` содержит логику, которая определяет, есть ли совпадение с каким-либо конкретным элементом, и я передаю его в качестве аргумента `FindIndex`. Вы можете передать любой метод с подходящей сигнатурой — `FindIndex` требует метод, который принимает экземпляр типа,

как у элемента массива, и возвращает `bool`. (Строго говоря, ему требуется `Predicate<T>`, который является своего рода делегатом, о чём я расскажу в главе 9.) Поскольку подойдет любой метод с нужной сигнатурой, мы можем делать свои критерии поиска такими простыми или сложными, как нам требуется.

Листинг 5.12. Поиск с `FindIndex`

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
    => Array.FindIndex(bins, IsNonZero);

private static bool IsNonZero(int value) => value != 0;
```

Кстати, логика для этого конкретного примера настолько проста, что написание отдельного метода для условия, очевидно, излишне. Для таких простых случаев, как эти, вы почти наверняка используете лямбда-синтаксис (используя `=>`, чтобы указать, что выражение представляет собой встроенную функцию). Об этом я тоже расскажу в главе 9, так что мы забегаем вперед, но я просто покажу, как выглядит эта более краткая запись. Код из листинга 5.13 делает абсолютно то же самое, что и код из листинга 5.12, но не требует, чтобы мы явно объявляли и писали целый дополнительный метод. (И на момент написания этого это еще и более эффективно, потому что в случае с лямбда-выражением компилятором генерируется код, который повторно использует созданный объект `Predicate<T>`, тогда как листинг 5.12 будет каждый раз создавать новый.)

Листинг 5.13. Использование лямбда-выражения с `FindIndex`

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
    => Array.FindIndex(bins, value => value != 0);
```

Как и в случае `IndexOf`, `FindIndex` предоставляет перегрузки, которые позволяют указать смещение, с которого начинается поиск, и количество элементов, которые необходимо проверить, прежде чем прервать поиск. Класс `Array` предоставляет и `FindLastIndex`, который работает в обратном направлении, — он соответствует `LastIndexOf`, так же как `FindIndex` соответствует `IndexOf`.

Когда вы ищете запись массива, которая соответствует определенным критериям, вас может не интересовать индекс соответствующего элемента. Вместо этого вам может потребоваться знать только значение первого соответствия. Очевидно, что его довольно легко получить: вы можете просто

использовать значение, возвращаемое `FindIndex` в сочетании с синтаксисом индекса массива. Однако в этом нет необходимости, поскольку класс `Array` предлагает методы `Find` и `FindLast`, которые выполняют поиск точно так же, как `FindIndex` и `FindLastIndex`, но возвращают первое или последнее совпадающее значение вместо индекса, по которому это значение было найдено.

Массив может содержать несколько элементов, соответствующих вашим критериям, и вы можете захотеть найти их все. Вы можете написать цикл, который вызывает `FindIndex`, добавляя его в индекс предыдущего совпадения, используя его в качестве начальной точки для следующего поиска и повторяя его, пока не достигнете конца массива или не получите результат `-1`, что говорит о том, что больше никаких совпадений не найдено. И это хороший способ, если вам нужно знать индекс каждого совпадения. Но если вас интересуют только знания всех совпадающих значений и вам не нужно точно знать, где эти значения расположены в массиве, вы можете использовать метод `FindAll`, показанный в листинге 5.14. Он сделает всю работу за вас.

Листинг 5.14. Поиск нескольких элементов с помощью `FindAll`

```
public T[] GetNonNullItems<T>(T[] items) where T : class  
    => Array.FindAll(items, value => value != null);
```

Листинг берет любой массив с элементами ссылочного типа и возвращает массив, содержащий только элементы, которые не содержат `null`.

Все методы поиска, которые я пока показал, проходят элементы массива по порядку, проверяя каждый элемент по очереди. Это работает достаточно неплохо, но с большими массивами может оказаться излишне дорогостоящим, особенно в случаях, когда сравнение является относительно сложным. Даже для простых сравнений, если иметь дело с массивами из миллионов элементов, такой вид поиска может занять достаточно много времени и вызвать видимые задержки. Однако мы можем проделать это намного эффективнее. Например, работая с массивом значений, отсортированных в порядке возрастания, на порядок лучшие результаты покажет бинарный поиск. В листинге 5.15 это показано.

Листинг 5.15. Производительность поиска и `BinarySearch`

```
var sw = new Stopwatch();  
  
int[] big = new int[100_000_000];  
Console.WriteLine("Initializing");  
sw.Start();
```

```
var r = new Random(0);
for (int i = 0; i < big.Length; ++i)
{
    big[i] = r.Next(big.Length);
}
sw.Stop();
Console.WriteLine(sw.Elapsed.ToString("s\\.f")); Console.WriteLine();

Console.WriteLine("Searching");
for (int i = 0; i < 6; ++i)
{
    int searchFor = r.Next(big.Length);
    sw.Reset();
    sw.Start();
    int index = Array.IndexOf(big, searchFor);
    sw.Stop();
    Console.WriteLine($"Index: {index}");
    Console.WriteLine($"Time: {sw.Elapsed:s\\.fffff}");
}
Console.WriteLine();

Console.WriteLine("Sorting");
sw.Reset();
sw.Start();
Array.Sort(big);
sw.Stop();
Console.WriteLine(sw.Elapsed.ToString("s\\.ff")); Console.WriteLine();

Console.WriteLine("Searching (binary)");
for (int i = 0; i < 6; ++i)
{
    int searchFor = r.Next() % big.Length;
    sw.Reset();
    sw.Start();
    int index = Array.BinarySearch(big, searchFor); sw.Stop();
    Console.WriteLine($"Index: {index}");
    Console.WriteLine($"Time: {sw.Elapsed:s\\.fffffff}");
}
```

В этом примере создается `int[]` со 100 000 000 значений. Он заполняется случайными числами с использованием класса `Random`, а затем используется `Array.IndexOf` для поиска в массиве некоторых случайно выбранных значений¹. Затем массив сортируется в порядке возрастания с использованием

¹ Я ограничил диапазон случайных чисел размером массива, потому что при полном диапазоне случайных чисел большинство попыток поиска будут неудачными.

`Array.Sort`. Это позволяет коду использовать метод `Array.BinarySearch` для поиска ряда случайно выбранных значений. Класс `Stopwatch` из пространства имен `System.Diagnostics` используется, чтобы измерить, сколько времени это занимает. (Странно выглядящий текст в последнем `Console.WriteLine` — это спецификатор формата, указывающий, сколько десятичных знаков мне нужно.) Измеряя такие микроскопические величины, мы ступаем на территорию под названием *микробенчмаркинг*. Измерение одной операции вне контекста может привести к вводящим в заблуждение результатам, потому что в реальных системах производительность зависит от множества факторов, которые взаимодействуют сложным, а иногда и непредсказуемым образом. Нужно смотреть на эти цифры скептически. Несмотря на это, масштаб различий в нашем случае довольно показателен. Вот что выводит моя система:

```
Initializing  
1.07
```

```
Searching  
Index: 55504605  
Time: 0.0191  
Index: 21891944  
Time: 0.0063  
Index: 56663763  
Time: 0.0173  
Index: 37441319  
Time: 0.0111  
Index: -1  
Time: 0.0314  
Index: 9344095
```

```
Time: 0.0032
```

```
Sorting  
7.3
```

```
Searching (binary)  
Index: 8990721  
Time: 0.0002616  
Index: 4404823  
Time: 0.0000205  
Index: 52683151  
Time: 0.0000019  
Index: -37241611  
Time: 0.0000018
```

```
Index: -49384544
Time: 0.0000019
Index: 88243160
Time: 0.000001
```

Для заполнения массива случайными числами требуется 1,07 секунды. (Большая часть этого времени уходит на генерацию чисел. Заполнение массива постоянным значением или счетчиком циклов занимает примерно 0,1 секунды.) Поиск в `IndexOf` занимает разное время. Самым медленным он был при отсутствии искомого значения — при неудачном поиске был получен индекс `-1`, и на него потребовалось 0,0314 секунды. Это потому, что `IndexOf` должен был просмотреть каждый элемент в массиве. В тех случаях, когда он находил совпадение, он работал быстрее, и скорость определялась тем, насколько рано он нашел совпадение. Самым быстрым в этом конкретном прогоне был поиск, при котором он нашел совпадение после чуть более 9 миллионов записей — это заняло 0,0032 секунды, что примерно в 10 раз быстрее, чем при необходимости просмотра всех 100 миллионов записей. Как и ожидалось, время увеличивается с количеством элементов, которые он вынужден проверять.

В среднем вы ожидаете, что успешный поиск займет примерно половину времени для наихудшего случая (при условии равномерно распределенных случайных чисел). Поэтому поиск будет занимать в среднем где-то около 0,016 секунды, а среднее значение будет зависеть от того, как часто он окажется безуспешным. Это еще не катастрофа, но уже вполне можно ожидать проблем. Что касается работы с пользовательским интерфейсом, то все, что занимает более 0,1 секунды, раздражает пользователя, поэтому, хотя наша средняя скорость может быть достаточно высокой, наш худший случай достаточно близок к пределу, когда уже нужно начинать беспокоиться. (И конечно, вы можете увидеть и худшие результаты на слабом оборудовании.) Хотя такая производительность не внушает особого беспокойства в случае сценариев, работающих на стороне клиента, она может оказаться серьезной проблемой на сильно загруженном сервере. Если вы будете проделывать такую большую работу для каждого входящего запроса, это серьезно ограничит число пользователей, которых может обслуживать каждый сервер.

Теперь посмотрите на время для бинарного поиска, техники, при которой не проверяется каждый элемент. Он начинается с элемента в середине массива. Если это требуемое значение, поиск может остановиться, но в противном

случае в зависимости от того, является ли найденное значение большим или меньшим, чем искомое, он может мгновенно узнать, в какой половине массива будет это значение (если оно вообще присутствует). Затем он переходит к середине оставшейся половины, и, если это неверное значение, он снова может определить, какая четверть будет содержать искомое значение. На каждом шаге поиск сокращается вдвое, а после нескольких сокращений в два раза он будет сведен к одному элементу. Если это не то значение, которое он ищет, то искомый элемент отсутствует.



Этот процесс объясняет необычные отрицательные числа, которые появляются в процессе бинарного поиска. Когда значение не найдено, процесс двоичного поиска завершится со значением, ближайшим к искомому, что может оказаться полезной информацией. Таким образом, отрицательное число все еще говорит нам, что поиск не удался, но это число является отрицательным индексом ближайшего значения.

Каждая итерация бинарного поиска сложнее, чем при простом линейном поиске, но при больших массивах она окупается, потому что таких итераций требуется гораздо меньше. В показанном примере необходимо выполнить только 27 шагов вместо 100 000 000. Очевидно, что с меньшими массивами преимущество уменьшается и есть некий минимальный размер массива, при котором относительная сложность бинарного поиска перевешивает выгоду. Если ваш массив содержит только 10 значений, линейный поиск вполне может оказаться быстрее. Но бинарный поиск — явный фаворит в случае со 100 000 000 элементов.

Значительно сокращая объем работы, `BinarySearch` работает намного быстрее, чем `IndexOf`. В примере наихудший вариант составляет 0,0002616 секунды (261,6 мкс), что примерно в 12 раз быстрее, чем лучший результат, который мы видели при линейном поиске. И этот первый поиск был необычайно медленным исключительным случаем; второй был на порядок быстрее, а остальные — на два порядка быстрее, т. е. достаточно быстрым, чтобы приблизиться к точке, в которой трудно выполнить точные измерения для отдельных операций². Таким образом, если код запущен и работает, все

² Более сложный тест показывает, что 261,1 мкс являются исключительным результатом: похоже, что первый поиск процессор выполняет относительно медленно. Это могут быть затраты, не связанные с поиском, который затрагивает только первый фрагмент кода, вызывающий `BinarySearch`, но относящиеся, например,

скорости поиска будут меньше 2 мкс. Самое интересное, что, когда не было найдено ни одного совпадения (что привело к отрицательному результату), `Array.IndexOf` показал самый медленный результат, но с `BinarySearch` случаи отсутствия совпадений выглядят довольно быстрыми: он определяет, что элемент отсутствует, в 15 000 раз быстрее, чем линейный поиск.

Помимо того что такой поиск потребляет гораздо меньше процессорного времени в каждом случае, он наносит меньший ущерб. Одну из наиболее коварных проблем с производительностью, которые могут возникнуть на современных компьютерах, преподносит код, который не только медленный сам по себе, но и вызывает замедление работы всего остального. Поиск `IndexOf` перетряхивает до 400 МБ данных для каждого неудачного поиска, и можно предположить, что в случае успешного поиска он пройдется в среднем по 200 МБ. Это приведет к очистке кэш-памяти ЦП, поэтому код и структуры данных, которые в противном случае могли бы остаться в быстрой кэш-памяти, придется еще раз извлекать из основной памяти, когда они потребуются; код, который использует `IndexOf` для такого большого массива, должен будет еще раз загрузить свое окружение обратно в кэш после завершения поиска. И если этот код на многопоточном сервере делит ЦП с другим кодом, он также может вытеснить из кэша данные других потоков, что также замедлит их работу. `BinarySearch` просматривает только несколько элементов массива, поэтому он будет оказывать минимальное влияние на кэш.

Есть только одна маленькая проблема: несмотря на то что отдельные поиски были намного быстрее, бинарный поиск в плане производительности в целом оказался полной катастрофой. Мы сэкономили почти одну десятую секунды на поиске, но, чтобы это сделать, пришлось потратить 7,3 секунды на сортировку массива. Бинарный поиск работает только для уже упорядоченных данных, и затраты на сортировку ваших данных могут перевесить преимущества. В этом примере потребуется выполнить около 500 поисков, прежде чем затраты на сортировку перевесит повышенная скорость поиска, и, конечно, это сработает, только если за это время не произойдет ничего, что заставит вас повторить сортировку. При настройке производительности всегда важно смотреть на весь сценарий целиком, а не только на микробенчмарки.

к JIT-компиляции. Когда интервалы становятся такими крошечными, вы достигаете предела, после которого микробенчмарк уже бесполезен.

Кстати, `Array.BinarySearch` предлагает перегрузки для поиска в некотором подразделе массива, аналогичные тем, которые мы видели для других методов поиска. Он также позволяет настроить логику сравнения. Это работает и с интерфейсами сравнения, которые я показал в предыдущих главах. По умолчанию он будет использовать `IComparable<T>`, реализация которого обеспечивается самими элементами массива, но вы можете предоставить собственный `IComparer<T>`. Метод `Array.Sort`, который я использовал для упорядочения элементов, также поддерживает сужение диапазона и использование собственной логики сравнения.

Существуют и другие методы поиска и сортировки, кроме тех, которые предоставляются самим классом `Array`. Все массивы реализуют `IEnumerable<T>` (где `T` — это тип элемента массива), что означает, что вы также можете использовать любые операции, предоставляемые функциональностью `LINQ to Objects .NET`. Это предлагает гораздо более широкий спектр функций поиска, сортировки, группировки, фильтрации и в целом работы с коллекциями объектов. Эти функции будут описаны в главе 10. Массивы поддерживаются в .NET больше, чем `LINQ`, что является одной из причин такого перекрытия функциональности, но, когда массивы содержат собственные эквиваленты стандартных операторов `LINQ`, версии массивов иногда могут оказаться более эффективными, поскольку `LINQ` является более обобщенным решением.

Многомерные массивы

Массивы, которые я показал до сих пор, были одномерными, но C# поддерживает две многомерные формы: ступенчатые массивы и прямоугольные массивы.

Ступенчатые массивы

Ступенчатый массив — это просто массив массивов. Существование такого рода массивов является естественным следствием того факта, что массивы имеют типы, отличные от типа их элементов. Поскольку `int[]` является типом, вы можете использовать его как тип элементов другого массива. Листинг 5.16 показывает синтаксис, который вполне ожидаем.

Опять же, я нарушил свое обычное правило для объявлений переменных. В обычных обстоятельствах я использовал бы `var` в первой строке, потому что тип очевиден из инициализатора, но я хотел показать синтаксис как объявления переменной, так и построения массива. А вот второй элемент

избыточности этого кода: при использовании синтаксиса инициализатора массива не нужно явно указывать размер, потому что компилятор подберет его сам. Я использовал эту функцию для вложенных массивов, но явно установил размер (5) для внешнего массива, чтобы показать, где размещается размер, потому что он может оказаться не там, где вы ожидаете.

Листинг 5.16. Создание ступенчатого массива

```
int[][] arrays = new int[5][]
{
    new[] { 1, 2 },
    new[] { 1, 2, 3, 4, 5, 6 },
    new[] { 1, 2, 4 },
    new[] { 1 },
    new[] { 1, 2, 3, 4, 5 }
};
```

Имя типа для ступенчатого массива задается достаточно просто. В целом, типы массивов имеют форму *ElementType*[], поэтому, если тип элемента **int**[], вполне ожидаемо, что результирующий тип массива будет выглядеть как **int**[][]], и это как раз то, что мы видим. Синтаксис конструктора более интересен. Он объявляет массив из пяти массивов, и на первый взгляд **new int[5][]** кажется вполне разумным способом записать это. Это согласуется с синтаксисом индекса массива для ступенчатых массивов; мы можем написать **arrays[1][3]**, что возвращает второй из этих пяти массивов, а затем извлекает из него четвертый элемент. (Между прочим, это не специализированный синтаксис — здесь нет нужды в специальной обработке, потому что за любым выражением, результатом которого является массив, может следовать индекс в квадратных скобках. Выражение **arrays[1]** возвращает массив **int**[], поэтому мы можем добавить к нему [3].)

А вот ключевое слово **new** обрабатывает ступенчатые массивы особым образом. В результате они выглядят совместимыми с синтаксисом доступа к элементам массива, но для этого приходится кое-что изменять. В случае одномерного массива шаблон для создания нового массива — это **new ElementType[length]**, поэтому для создания массива из пяти элементов нужно написать **new ElementType[5]**. Если то, что вы создаете, является набором массивов **int**, разве не будет разумным ожидать **int[]** вместо **ElementType**? Подразумевается, что синтаксис должен быть таким: **new int[][][5]**.

Выглядит логично, но неправильно, а все потому, что сам синтаксис типа массивов по сути перевернут. Массивы — это конструируемые типы, как

и обобщения. В случае обобщения имя обобщенного типа, из которого мы конструируем фактический тип, предшествует аргументу типа (например, `List<int>` берет обобщенный `List<T>` и конструирует его с аргументом типа `int`). Если бы у массивов был синтаксис обобщений, мы бы ожидали увидеть `array<int>` для одномерного массива, `array<array<int>>` — для двух измерений и т. д. Тип элемента был бы указан *после* той части, которая означает, что мы хотим получить массив. Но типы массивов делают это наоборот — признак массива указывается символами `[]`, поэтому тип элемента стоит первым. Вот почему гипотетический логически правильный синтаксис построения массива выглядит странно. С# избегает странностей тем, что не делает чрезмерного упора на логику, а вместо этого просто размещает размер там, где большинство людей ожидают его увидеть, а не там, где он обоснованно должен быть.



В C# не определены конкретные ограничения на число измерений, но есть зависящие от реализации ограничения во время выполнения. (Компилятор Microsoft даже не поморщился, когда я затребовал 5000-мерный ступенчатый массив, но CLR отказался загружать итоговую программу. Фактически он не будет загружать что-либо с более чем 4144 измерениями, а некоторые проблемы с производительностью возникли уже при 2000 измерений.) Синтаксис расширяется вполне очевидным образом — например, `int[][][]` для типа и `new int[5][][]` — для конструирования.

Листинг 5.16 инициализирует массив пятью одномерными массивами `int[]`. Компоновка кода должна прояснить, почему этот вид массива называется ступенчатым: каждая строка имеет разную длину. Для массива массивов не требуется прямоугольная компоновка. Я могу зайти еще дальше. Массивы являются ссылочными типами, поэтому я мог бы установить для некоторых строк значение `NULL`. Если бы я отказался от синтаксиса инициализатора массива и инициализировал элементы массива по отдельности, я мог бы сделать так, что некоторые одномерные массивы `int[]` появились бы в более чем одной строке.

Поскольку каждая строка в этом ступенчатом массиве содержит массив, я получил шесть объектов — пять массивов `int[]`, а затем массив `int[][][]`, который содержит ссылки на них. Если вы добавите больше измерений, вы получите еще больше массивов. Для определенных видов работ непрямоугольность и большое количество объектов могут оказаться проблемой, поэтому С# поддерживает и другой тип многомерного массива.

Прямоугольные массивы

Прямоугольный массив – это один объект массива, который поддерживает многомерное индексирование. Если бы C# не предлагал многомерные массивы, мы могли бы создать нечто похожее на них, используя соглашение. Если вам нужен массив из 10 строк и 5 столбцов, вы можете создать одномерный массив из 50 элементов, а затем использовать для доступа код, подобный `myArray [i + (5 * j)]`, где `i` – индекс столбца и `j` – индекс строки. Это был бы массив, который вы рассматриваете как двумерный, хотя на самом деле это всего лишь один большой непрерывный блок. Прямоугольный массив – это по сути та же идея, но C# делает всю работу за вас. В листинге 5.17 показано, как объявлять и создавать прямоугольные массивы.



Прямоугольные массивы – это не просто удобно. Не стоит забывать и о безопасности типов: `int [,]` – это тип, отличный от `int []` или `int [,,]`, поэтому, если вы напишите метод, который ожидает двумерный прямоугольный массив, C# не пропустит ничего другого.

Листинг 5.17. Прямоугольные массивы

```
int[,] grid = new int[5, 10];
var smallerGrid = new int[,]
{
    {1,2,3,4},
    {2,3,4,5},
    {3,4,5,6},
};
```

Как видите, имена типов прямоугольных массивов используют только одну пару квадратных скобок независимо от того, сколько у них измерений. Число запятых внутри скобок обозначает количество измерений, поэтому эти примеры с одной запятой являются двумерными.

Синтаксис инициализатора очень похож на синтаксис многомерных массивов (см. листинг 5.16), за исключением того, что я не начинаю каждую строку с `new[]`, потому что все это – один большой массив, а не массив массивов. Числа в листинге 5.17 формируют фигуру, которая является явно прямоугольной, и, если вы попытаетесь превратить ее в ступенчатую (изменив размер строк), компилятор сообщит об ошибке. Это распространяется на более высокие измерения. Если вам нужен трехмерный «прямоугольный» массив, он должен быть кубоидом. В листинге 5.18 показан кубоидный

массив. Вы можете представить инициализатор как список из двух прямоугольных разрезов, составляющих кубоид. Следующая ступень — это гиперкубондные массивы (хотя они все еще называются прямоугольными массивами независимо от того, сколько в них измерений).

Листинг 5.18. Кубоидный «прямоугольный» массив размером $2 \times 3 \times 5$

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

Синтаксис для доступа к прямоугольным массивам достаточно предсказуем. Если вторая переменная из листинга 5.17 находится в области видимости, мы могли бы написать `smallerGrid[2, 3]` для доступа к последнему элементу в массиве; как и в случае одномерных массивов, индексы начинаются с нуля, так что речь идет о четвертом элементе третьей строки.

Помните, что свойство `Length` массива возвращает общее количество элементов в массиве. Поскольку прямоугольные массивы содержат все элементы в одном массиве (а не являются массивами, которые ссылаются на другие массивы), результатом будет произведение размеров всех измерений. Например, в случае прямоугольного массива из 5 строк и 10 столбцов свойство `Length` будет равно 50. Если вы во время выполнения хотите получить размер определенного измерения, используйте метод `GetLength`, который принимает один аргумент `int`, указывающий измерение, размер которого вы хотите узнать.

Копирование и изменение размера

Иногда вам захочется перемещать фрагменты данных из массива в массив. Возможно, вы захотите вставить элемент в середину массива, переместив следующие элементы на одну позицию (и потерять один элемент в конце,

поскольку размеры массива фиксированы). Или вы можете захотеть переместить данные из одного массива в другой, возможно, другого размера.

Статический метод `Array.Copy` принимает ссылки на два массива вместе с числом, указывающим, сколько элементов нужно скопировать. Он предлагает перегрузки, так что вы можете указать позиции в двух массивах, с которых нужно начать копирование. (Самая простая перегрузка начинает с первого элемента каждого массива.) Вам разрешено передавать один и тот же массив в качестве источника и цели копирования, и такое перекрытие будет обработано правильно: копирование работает так, как если бы все элементы были сначала скопированы во временное местоположение, а только после этого записывались в целевой массив.



Помимо статического метода `Copy` класс `Array` определяет нестатический метод `CopyTo`, который копирует весь массив в целевой массив, начиная с указанного смещения. Этот метод есть, потому что все массивы реализуют определенные интерфейсы коллекций, включая `ICollection<T>` (где `T` – тип элемента массива), который и определяет метод `CopyTo`. `CopyTo` не гарантирует корректную обработку перекрытия при копировании массива в себя, и документация рекомендует использовать `Array.Copy` в сценариях, когда вы знаете, что будете иметь дело с массивами. `CopyTo` используется только для общего кода, который может работать с любой реализацией интерфейса коллекции.

Копирование элементов из одного массива в другой может оказаться необходимым, когда вы имеете дело с переменным количеством данных. Обычно вы создаете массив больше, чем нужно изначально, но если и он в итоге заполнится, потребуется новый, больший массив, в который нужно будет скопировать содержимое старого. На самом деле в случае одномерных массивов класс `Array` может сделать это за вас с помощью метода `Resize`. Имя метода немного вводит в заблуждение, потому что размеры массивов не могут быть изменены, так что он выделяет новый массив и копирует в него данные из старого. `Resize` может создать массив большего или меньшего размера, и, если вы попросите его уменьшить размер, он просто скопирует столько элементов, сколько поместится.

Говоря о методах, которые копируют данные массива, я должен упомянуть `Reverse`, который просто меняет порядок элементов массива. Кроме того, хотя речь идет не только о копировании, метод `Array.Clear` часто полезен

в сценариях, где вы манипулируете размерами массива. Он позволяет сбросить некоторый диапазон массива до его начального нулевого состояния.

Эти методы перемещения данных внутри массивов полезны для построения более гибких структур данных поверх базовых функций, предлагаемых массивами. Но вам зачастую не придется использовать их самостоятельно, потому что библиотека классов предоставляет несколько полезных классов коллекций, которые делают это за вас.

Класс `List<T>`

`List<T>` — это класс, определенный в пространстве имен `System.Collections.Generic`, который содержит последовательность переменной длины из элементов типа `T`. Он предоставляет индексатор, который позволяет получать и устанавливать элементы согласно позиции, поэтому `List<T>` ведет себя как массив изменяемого размера. Они не полностью взаимозаменямы, потому что вы не можете передать `List<T>` в качестве аргумента для параметра, который ожидает массив `T[]`, но и массивы, и `List<T>` реализуют ряд общих обобщенных интерфейсов коллекций, которые мы рассмотрим позже. Например, если вы пишете метод, который принимает `IList<T>`, он сможет работать с массивом или `List<T>`.



Хотя код с использованием индексатора напоминает доступ к элементу массива, это не совсем то же самое. Индексатор является своего рода свойством, поэтому он имеет те же проблемы с изменяемыми значимыми типами, о которых я говорил в главе 3. Для переменной `pointList` типа `List <Point>` (где `Point` — это изменяемый значимый тип в пространстве имен `System.Windows`) вы не можете написать `pointList[2].X = 2`, поскольку `pointList[2]` возвращает копию значения и код фактически запрашивает изменение этой временной копии. Обновление было бы потеряно, поэтому C# запрещает подобное. Но это работает с массивами. Если `pointArray` имеет тип `Point[]`, `pointArray[2]` не получает элемент, а идентифицирует его, позволяя на месте изменить значение элемента массива, написав `pointArray[2].X = 2`. (Поскольку в C# 7.0 были добавлены возвращаемые значения `ref`, стало возможным писать индексаторы, которые работают именно таким образом, но `List<T>` и `IList<T>` были созданы задолго до этого.) С неизменяемыми значимыми типами, такими как `Complex`, это различие под вопросом, потому что вы никогда не сможете изменить их значения на месте — вам придется перезаписать элемент новым значением, будь то массив или список.

В отличие от массива, List<T> содержит методы, которые изменяют его размер. Метод Add добавляет новый элемент в конец списка, а AddRange может добавить сразу несколько. Insert и InsertRange добавляют элементы в любой точке, перемещая все элементы после точки вставки вниз, чтобы освободить место. Все эти четыре метода делают список длиннее, но List<T> также предоставляет метод Remove, который удаляет первый экземпляр с указанным значением; RemoveAt, который удаляет элемент в определенной позиции; и RemoveRange, который удаляет несколько элементов, начиная с определенно-го индекса. Все эти методы сдвигают элементы назад, закрывая зазор, оставленный удаленным элементом или элементами, что делает список короче.



Для хранения своих элементов List<T> использует массив. Это означает, что все элементы расположены в одном блоке памяти и он хранит их смежно. Это делает нормальный доступ к элементам очень эффективным, но именно поэтому вставка должна смещать элементы вверх, чтобы освободить место для нового элемента, а удаление — вниз, чтобы скратить разрыв.

Листинг 5.19 показывает, как создать List<T>. Это обычный класс, поэтому мы используем нормальный синтаксис конструктора. Пример показывает, как добавлять и удалять записи, а также как получить доступ к элементам с использованием синтаксиса индексатора, похожего на используемый в массиве. Он также демонстрирует, что List<T> хранит свой размер в свойстве Count, которое, по-видимому, намеренно отличается от Length массива. (На самом деле массивы тоже содержат Count, потому что реализуют ICollection и ICollection<T>. Однако они используют явную реализацию интерфейса, что означает, что вы можете увидеть свойство Count массива только через ссылку на один из этих типов интерфейса.)

Листинг 5.19. Использование List<T>

```
var numbers = new List<int>();
numbers.Add(123);
numbers.Add(99);
numbers.Add(42);
Console.WriteLine(numbers.Count);
Console.WriteLine($"{numbers[0]}, {numbers[1]}, {numbers[2]}");

numbers[1] += 1;
Console.WriteLine(numbers[1]);
```

```
numbers.RemoveAt(1);
Console.WriteLine(numbers.Count);
Console.WriteLine($"{numbers[0]}, { numbers[1]}");
```

Из-за того что `List<T>` может увеличиваться и уменьшаться по мере необходимости, не нужно указывать его размер при конструировании. Но если хотите, то можете указать его емкость. Емкость списка — это объем места, который он в настоящее время занимает для хранения элементов, и он, как правило, будет отличаться от количества содержащихся элементов. Чтобы избежать выделения нового внутреннего массива каждый раз, когда вы добавляете или удаляете элемент, он независимо от размера массива отслеживает количество используемых элементов. Когда ему требуется больше места, он перераспределяет место, создавая новый массив, который будет больше, чем нужно, на коэффициент, пропорциональный размеру. Это означает, что если ваша программа многократно добавляет элементы в список, то чем больше он становится, тем реже нужно выделять новый массив, а доля резервной емкости после каждого перераспределения будет оставаться примерно одинаковой.

Если вы заранее знаете, что в конечном итоге будете хранить определенное количество элементов в списке, вы можете передать это число конструктору и он выделит именно столько емкости, что означает, что дальнейшее перераспределение не потребуется. Нужно понимать, что это не приведет к ошибкам — вы просто запрашиваете начальную емкость и позже можете передумать.

Если вас беспокоит мысль о неиспользуемой памяти в списке, но вы изначально не знаете точно, сколько места потребуется, то вызовете метод `TrimExcess`, как только поймете, что список заполнен. Он перераспределит место во внутреннем хранилище, оставив достаточно, чтобы вместить текущее содержимое списка и ничего лишнего. Это не всегда обирачивается выгодой. Чтобы гарантировать, что он точно использует нужный объем пространства, `TrimExcess` должен создать новый массив правильного размера, оставив старый и слишком большой массив сборщику мусора. В некоторых случаях расходы на принудительное дополнительное распределение только для того, чтобы урезать размер, могут быть выше, чем издержки, связанные с наличием неиспользованной емкости.

У списков есть и третий конструктор. Помимо конструктора по умолчанию и того, который задает емкость, вы также можете передать набор данных для инициализации списка. Вы можете передать любой `IEnumerable<T>`.

Вы можете предоставить для списка начальное содержимое с помощью синтаксиса, похожего на инициализатор массива. Листинг 5.20 помещает в новый список те же три значения, что мы видим в начале листинга 5.19. Это единственная форма; в отличие от массивов, вы не можете опустить `new List<int>`, когда объявление переменной явно содержит тип (т. е. когда вы не используете `var`). Кроме того, компилятор не будет определять аргумент типа, поэтому, в то время как в случае массива вы можете написать просто `new []`, за которым следует инициализатор, нельзя написать `new List<>`.

Листинг 5.20. Инициализатор списка

```
var numbers = new List<int> { 123, 99, 42 };
```

Это будет скомпилировано в код, который вызывает `Add` по разу для каждого элемента в списке. Вы можете использовать этот синтаксис с любым типом, который имеет подходящий метод `Add` и реализует интерфейс `IEnumerable`. Это работает, даже если `Add` является методом расширения. (Таким образом, если какой-либо тип реализует `IEnumerable`, но не предоставляет метод `Add`, вы можете использовать этот синтаксис инициализатора, если предоставите свой собственный `Add`.)

`List<T>` содержит методы `IndexOf`, `LastIndexOf`, `Find`, `FindLast`, `FindAll`, `Sort` и `Binary Search` для поиска и сортировки элементов списка. Они делают то же самое, что и их тезки из массивов, хотя `List<T>` предоставляет их в качестве методов экземпляра, а не статичных.

Итак, мы разобрали два способа задания списка значений: массивы и списки. К счастью, интерфейсы позволяют писать код, который может работать с любым из них, поэтому вам не нужно писать два набора функций, если вы хотите поддерживать и списки, и массивы.

Интерфейсы списков и последовательностей

Библиотека классов .NET определяет несколько интерфейсов, представляющих собой коллекции. Три из них относятся к простым линейным последовательностям, которые вы можете сохранить в массиве или списке: `IList<T>`, `ICollection<T>` и `IEnumerable<T>`. Все они расположены в пространстве имен `System.Collections.Generics`.

Их три, потому что разный код предъявляет разные требования. Некоторым методам необходим произвольный доступ к любому пронумерованному

элементу в коллекции, но не все коллекции это поддерживают — некоторые последовательности способны выдавать элементы только друг за другом, и может оказаться невозможно перепрыгнуть прямо к n -му элементу. Рассмотрим, например, последовательность, хранящую нажатия клавиш, — каждый элемент появляется в ней только тогда, когда пользователь нажимает следующую клавишу. Чем менее требователен интерфейс, тем с более широким диапазоном источников может работать ваш код.

`IEnumerable<T>` является наиболее общим из интерфейсов коллекции, потому что его требования к разработчику минимальны. Я уже несколько раз упоминал о нем, потому что это важный интерфейс, который часто используется, но до сих пор не показал его определение. Как показано в листинге 5.21, он объявляет только один метод.

Листинг 5.21. `IEnumerable<T>` и `IEnumerable`

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Из-за использования наследования `IEnumerable<T>` требует от разработчиков реализации еще и `IEnumerable`, который выглядит почти идентичным. Это необобщенная версия `IEnumerable<T>`, и его метод `GetEnumerator`, как правило, не делает ничего, кроме вызова обобщенной реализации. Причина, по которой у нас теперь есть обе формы, заключается в том, что необобщенный `IEnumerable` существует еще с .NET v1.0, где не поддерживались обобщения. Появление обобщений в .NET v2.0 позволило более точно реализовать задумку, стоящую за `IEnumerable`, но старый интерфейс должен был остаться для совместимости. Таким образом, эти два интерфейса фактически требуют одного и того же: метода, который возвращает перечислитель. Так что же такое перечислитель? Листинг 5.22 показывает как обобщенный, так и необобщенный интерфейс.

Модель использования `IEnumerable<T>` (а также `IEnumerable`) заключается в том, что вы вызываете `GetEnumerator` для получения перечислителя, который можно использовать для перебора всех элементов коллекции. Для

этого вы вызываете `MoveNext()` перечислителя, и если он возвращает `false`, то это означает, что коллекция пуста. В противном случае свойство `Current` теперь будет содержать первый элемент коллекции. Далее вы снова вызываете `MoveNext()` и переходите к следующему элементу. До тех пор пока он продолжает возвращать `true`, следующий элемент доступен через `Current`. (Метод `Reset` – это исторический артефакт, добавленный для обеспечения совместимости с COM, межязыковой объектной моделью Windows до .NET. Документация позволяет реализациям выбрасывать из `Reset` `NotSupportedException`, поэтому обычно вы не будете пользоваться этим методом.)

Листинг 5.22. `IEnumerator<T>` и `IEnumerator`

```
public interface IEnumarator<out T> : IDisposable, IEnumarator
{
    T Current { get; }
}

public interface IEnumarator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```



Обратите внимание, что реализации `IEnumarator<T>` необходимы для реализации `IDisposable`. Вы должны вызывать `Dispose` для перечислителей, как только закончите с ними, и многие из них опираются на этот принцип.

Цикл `foreach` в C# выполняет всю работу, необходимую для прохода по перечислимой коллекции, включая генерацию кода, который вызывает `Dispose`, даже если цикл завершается досрочно из-за оператора `break`, ошибки или, боже упаси, оператора `goto`³. В главе 7 мы поговорим об использовании `IDisposable` более подробно.

³ Удивительно, но `foreach` не требует какого-либо определенного интерфейса; он будет использовать все что угодно с методом `GetEnumerator`, который возвращает объект, предоставляющий метод `MoveNext` и свойство `Current`. Это историческая хитрость – еще до обобщений это было единственным способом производить перебор коллекций элементов с типизированными значениями без использования упаковки. Глава 7 описывает упаковку.

`IEnumerable<T>` лежит в основе LINQ to Objects, о чем я расскажу в главе 10. Операторы LINQ доступны для любого объекта, который реализует этот интерфейс.

В .NET Core 3.0 и .NET Standard 2.1 добавлен новый интерфейс `IAsyncEnumerable<T>`. Концептуально он идентичен `IEnumerable<T>`, предоставляя возможность доступа к последовательности элементов. Разница в том, что он поддерживает асинхронную работу. Как показано в листинге 5.23, он и его двойник `IAsyncEnumerator<T>` напоминают `IEnumerable<T>` и `IEnumerator<T>`. Основным отличием является использование функционала асинхронного программирования `ValueTask<T>` и `CancellationToken`, что будет описано в главе 16. Есть также и ряд незначительных отличий: не существует необобщенных версий этих интерфейсов, а также отсутствует возможностьброса существующего асинхронного перечислителя (хотя, как отмечалось ранее, многие синхронные перечислители генерируют исключение `NotSupportedException`, если вы вызываете `Reset`).

Листинг 5.23. `IAsyncEnumerable<T>` и `IAsyncEnumerator<T>`

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(
        CancellationToken cancellationToken = default);
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }

    ValueTask<bool> MoveNextAsync();
}
```

Вы можете использовать `IAsyncEnumerable<T>` со специальной формой цикла `foreach`, в которой вы добавляете префикс к ключевому слову `await`. Это можно использовать только в методе, помеченном ключевым словом `async`. Глава 17 подробно описывает ключевые слова `async` и `await`, а также использование `await foreach`.

Хотя `IEnumerable<T>` важен и достаточно широко используется, он довольно ограничен. Все, что вы можете, — это запрашивать элементы один за другим, а он будет выдавать их в том порядке, который считает нужным. Он не дает возможности изменить коллекцию или даже узнать, сколько элементов она

содержит, без необходимости перебирать ее всю. Для этих задач у нас есть `ICollection<T>`, что показано в листинге 5.24.

Листинг 5.24. `ICollection<T>`

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool Remove(T item);

    int Count { get; }
    bool IsReadOnly { get; }
}
```

Его использование требует от разработчиков реализации также и `IEnumerable<T>`, но обратите внимание, что отсутствует наследование необобщенной коллекции `ICollection`. Такой интерфейс тоже есть, но он представляет собой другую абстракцию: в нем отсутствуют все методы, кроме `CopyTo`. Представляя обобщения, компания Microsoft изучила то, как использовались необобщенные типы коллекций, и пришла к выводу, что один дополнительный метод, добавленный к старому `ICollection`, не сделает интерфейс заметно полезнее, чем `IEnumerable`. Хуже того, он также включал свойство `SyncRoot`, предназначеннное для некоторых многопоточных сценариев, но на практике оказавшееся плохим решением. Таким образом, абстракция, представленная `ICollection`, не получила обобщенного эквивалента, о чём никто особенно не жалел. В ходе изучения Microsoft также обнаружила проблему отсутствия обобщенного интерфейса для изменяемых коллекций и поэтому подогнала под эти цели `ICollection<T>`. Было не совсем правильным прикреплять старое имя к другой абстракции, но поскольку почти никто не использовал старую необобщенную коллекцию `ICollection`, к особым проблемам это не привело.

Третий интерфейс для последовательных коллекций — это `IList<T>`, и все типы, которые реализуют его, должны реализовывать `ICollection<T>` и, следовательно, `IEnumerable<T>`. Как и следовало ожидать, `List<T>` реализует `IList<T>`. Массивы также его реализуют, используя свой тип элементов в качестве аргумента для `T`. Листинг 5.25 показывает, как выглядит интерфейс.

Листинг 5.25. IList<T>

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IComparable<T>
{
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);

    T this[int index] { get; set; }
}
```

Опять же, хотя существует необобщенный `IList`, данный интерфейс не имеет к нему прямого отношения, даже если и представляет собой схожую концепцию. Необобщенный `IList` содержит эквиваленты членов `IList<T>`, а также эквиваленты для большей части `ICollection<T>`, включая отсутствующие в `ICollection`. Так что вполне можно было бы требовать реализаций `IList<T>` для того, чтобы реализовать `IList`, но это заставило бы реализаций иметь две версии большинства членов, одна из которых работала бы с параметром типа `T`, а другая с `object`, потому что именно `object` использовали старые необобщенные интерфейсы. Это также заставило бы коллекции предоставлять неиспользуемое свойство `SyncRoot`. Преимущества не способны перевесить эти неудобства, и поэтому реализации `IList<T>` не обязаны реализовывать еще и `IList`. Это возможно, и `List<T>` так и делает, но выбор зависит от конкретного класса коллекции.

Неудачное следствие того, как связаны эти три обобщенных интерфейса, заключается в том, что они не содержат абстракций, которые представляют собой пронумерованные коллекции только для чтения или даже коллекции фиксированного размера. В то время как `IEnumerable<T>` — это абстракция только для чтения, она упорядочена и не дает возможности перейти непосредственно к n -му значению. До версии .NET 4.5 (в которой были представлены различные новые интерфейсы коллекций) единственным вариантом для индексированного доступа был `IList<T>`, но он также определяет методы для вставки и индексированного удаления, а также подразумевает обязательное внедрение `ICollection<T>` с добавлением и основанными на значении методами удаления. Вы, возможно, гадаете, как массивы могут реализовывать эти интерфейсы, учитывая, что все они имеют фиксированный размер.

Массивы сглаживают эту проблему, используя явную реализацию интерфейса, чтобы скрыть методы `IList<T>`, которые способны изменить длину списка, что препятствует их использованию. (Как вы видели в главе 3, эта

техника позволяет вам обеспечить полную реализацию интерфейса, но при этом оставляет за вами выбор того, какие члены являются непосредственно видимыми.) Тем не менее вы можете сохранить ссылку на массив в переменной типа `IList<T>`, делая эти методы видимыми. Листинг 5.26 использует это для вызова метода массива `IList<T>.Add`. Однако это приводит к ошибке во время выполнения.

Листинг 5.26. Попытка (неудачная) увеличить массив

```
IList<int> array = new[] { 1, 2, 3 };
array.Add(4); // Будет сгенерировано исключение
```

Метод `Add` вызывает исключение `NotSupportedException` с сообщением об ошибке, которое утверждает, что коллекция имеет фиксированный размер. Если вы заглянете в документацию для `IList<T>` и `ICollection<T>`, то увидите, что всем членам, которые могли бы изменить коллекцию, разрешается выдавать эту ошибку. Во время выполнения вы посредством свойства интерфейса `IsReadOnly` можете узнать, будет ли это происходить для всех модификаций `ICollection<T>`. Тем не менее это свойство не поможет вам, когда коллекция допускает только определенные изменения. (Например, размер массива фиксирован, но вы все равно можете изменять элементы.)

Это приводит к раздражающей проблеме: если вы пишете код, который на самом деле требует изменяемой коллекции, нет способа об этом объявить. Если метод принимает `IList<T>`, трудно понять, будет ли он пытаться изменить размер списка. Несовпадения приводят к исключениям во время выполнения, и эти исключения вполне могут появиться в коде, который не делает ничего плохого, а ошибку допустила вызывающая сторона, передав не тот вид коллекции. Эти проблемы не приводят к неработоспособности; в динамически типизированных языках эта степень неопределенности во время компиляции фактически является нормой и никак не мешает писать хороший код.

Существует класс `ReadOnlyCollection<T>`, но, как мы увидим позже, он решает другую проблему. Это класс-оболочка, а не интерфейс, поэтому существует много всего, что представляет собой коллекции фиксированного размера, не являясь при этом `ReadOnlyCollection<T>`. Если потребуется написать метод с параметром типа `ReadOnlyCollection<T>`, он не сможет напрямую работать с определенными типами коллекций (включая массивы). В любом случае это даже не та же самая абстракция, потому что «только для чтения» — это более жесткое ограничение, чем фиксированный размер.

.NET определяет `IReadOnlyList<T>`, интерфейс, который предоставляет лучшее решение для индексируемых коллекций только для чтения (хотя он по-прежнему бесполезен в случае с модифицируемыми коллекциями фиксированного размера). Как и `IList<T>`, ему требуется реализация `IEnumerable<T>`, но не требуется `ICollection<T>`. Он определяет два члена: `Count`, который возвращает размер коллекции (так же, как `ICollection<T>.Count`) и индексатор только для чтения. Это решает большинство проблем, связанных с использованием `IList<T>` для коллекций только для чтения. Одна небольшая проблема заключается в том, что, поскольку он более новый, чем большинство других описанных здесь интерфейсов, он не везде поддерживается. (Он был введен в .NET 4.5 в 2012 году, через семь лет после `IList<T>`.) Так что если вы столкнетесь с API, который требует `IReadOnlyList<T>`, то можете быть уверены, что он не будет пытаться изменить коллекцию. Если же API требует `IList<T>`, то становится трудно понять, то ли он собирается изменить коллекцию, то ли был написан до изобретения `IReadOnlyList<T>`.



Коллекции не должны быть доступны только для чтения, чтобы реализовать `IReadOnlyList<T>` — изменяемый список вполне может иметь фасад списка только для чтения. Таким образом, этот интерфейс реализован всеми массивами, а также `List<T>`.

Проблемы и интерфейсы, которые я только что обсудил, поднимают вопрос: какой же тип нужно использовать при написании кода или классов, которые работают с коллекциями? Как правило, вы получаете наибольшую гибкость, если вашему API для работы требуется наименее конкретный тип. Например, если вам достаточно `IEnumerable<T>`, не нужно требовать `IList<T>`. Аналогично интерфейсы обычно лучше конкретных типов, поэтому вы должны предпочтеть `IList<T>`, — как `List<T>`, так и `T[]`. Иногда вопрос производительности обуславливает использование более определенного типа; если у вас есть цикл, критически влияющий на общую производительность вашего приложения и работающий с содержимым коллекции, вы можете обнаружить, что код выполняется быстрее, если он работает только с типами массивов, поскольку CLR может выполнять более эффективную оптимизацию, когда точно знает, чего ожидать. Но во многих случаях разница будет слишком мала, чтобы ее можно было измерить, и не будет оправдывать неудобство привязки к конкретной реализации. Поэтому никогда не следует

предпринимать такой шаг без измерений производительности выполняемой задачи и оценки выгода. (И если вы всерьез рассматриваете такое ориентированное на производительность изменение, вам также следует взглянуть на методы, описанные в главе 18.) Если вы обнаружите возможный выигрыш в производительности, но при этом пишете общую библиотеку, в которой вы хотите обеспечить как гибкость, так и наилучшую производительность, есть несколько способов добиться и того и другого. Вы можете использовать перегрузки, чтобы вызывающая сторона могла передавать либо интерфейс, либо определенный тип. Или вы могли бы написать один открытый метод, который принимает интерфейс, но проверяет его на известные типы и выбирает между различными ветками внутреннего кода на основе переданного вызывающей стороной.

Интерфейсы, которые мы только что рассмотрели, не являются единственными обобщенными интерфейсами коллекций, потому что простые линейные списки — не единственный вид коллекции. Но прежде, чем перейти к остальным, я хочу подойти к перечислениям и спискам с обратной стороны и показать, как мы реализуем эти интерфейсы.

Реализация списков и последовательностей

Часто бывает полезно предоставить информацию в виде `IEnumerable<T>` или `IList<T>`. Первое особенно важно, потому что .NET предоставляет мощный инструментарий для работы с последовательностями в форме `LINQ to Objects`, который я покажу в главе 10. `LINQ to Objects` предоставляет различные операторы, все из которых работают в терминах `IEnumerable<T>`. `IList<T>` — это полезная абстракция, если требуется произвольный доступ к любому элементу по индексу. Некоторые библиотеки классов ожидают `IList<T>`. Иногда вам требуется привязка коллекции объектов к какому-либо элементу управления списком. Так, например, некоторые библиотеки пользовательского интерфейса могут запрашивать как `IList`, так и `IList<T>`.

Вы можете реализовать эти интерфейсы вручную, поскольку ни один из них не является особенно сложным. Однако в данном случае могут пригодиться C# и библиотека классов .NET. В них существует прямая языковая поддержка реализации `IEnumerable<T>`, а также поддержка библиотеки классов для обобщенных и необобщенных интерфейсов списков.

Реализация `IEnumerable<T>` с итераторами

C# поддерживает специальную форму метода, называемую *итератор*. Итератор — это метод, который создает перечислимые последовательности, используя специальное ключевое слово `yield`. Листинг 5.27 показывает простой итератор и код, который его использует. Итератор отображает числа в обратном порядке, от 5 до 1.

Листинг 5.27. Простой итератор

```
public static IEnumerable<int> Countdown(int start, int end)
{
    for (int i = start; i >= end; --i)
    {
        yield return i;
    }
}

private static void Main(string[] args)
{
    foreach (int i in Countdown(5, 1))
    {
        Console.WriteLine(i);
    }
}
```

Итератор очень похож на обычный метод, но способ, которым он возвращает значения, отличается. Итератор в листинге 5.27 имеет тип возвращаемого значения `IEnumerable<int>`, и все же он, похоже, не возвращает ничего подобного. Вместо обычного оператора `return` он использует оператор `yield return`, который возвращает единственное целое число, а не коллекцию. Итераторы выдают значения по одному с помощью операторов `yield return`, и, в отличие от обычного `return`, метод может продолжать выполняться после возврата значения. Он завершается, только когда выполняется до конца, досрочно останавливается с помощью оператора `yield break` или вызывает исключение. Листинг 5.28 показывает это довольно наглядно. Каждый `yield return` приводит к возврату значения из последовательности, поэтому в нашем случае получаются числа 1–3.

Хотя это довольно просто в теории, способ работы итератора довольно сложен, потому что код в итераторах выполняется не так, как остальной код. Вспомните, что в случае с `IEnumerable<T>` за получение очередного значения отвечает вызывающая сторона; цикл `foreach` получит перечислитель, а за-

тем будет повторно вызывать `MoveNext()` до тех пор, пока он не вернет `false`, при этом ожидая, что свойство `Current` содержит текущее значение. Итак, как же листинги 5.27 и 5.28 вписываются в эту модель? Вы можете подумать, что, возможно, C# сохраняет все значения, которые выдает итератор, в `List<T>` и возвращает его после завершения итератора, но легко увидеть, что это не так, написав итератор, который никогда не завершается, такой как в листинге 5.29.

Листинг 5.28. Очень простой итератор

```
public static IEnumerable<int> ThreeNumbers()
{
    yield return 1;
    yield return 2;
    yield return 3;
}
```

Листинг 5.29. Бесконечный итератор

```
public static IEnumerable<BigInteger> Fibonacci()
{
    BigInteger v1 = 1;
    BigInteger v2 = 2;

    while (true)
    {
        yield return v1;
        var tmp = v2;
        v2 = v1 + v2;
        v1 = tmp;
    }
}
```

Этот итератор работает бесконечно; у него есть цикл `while` с условием `true`, в нем нет оператора `break`, и поэтому итератор никогда не остановится по своей воле. Если бы C# пытался выполнить итератор до конца, прежде чем что-либо вернуть, он бы застрял здесь. (Числа все возрастают, поэтому, если он будет работать достаточно долго, метод все-таки прекратит работу, выдав исключение `OutOfMemoryException`, но он никогда не вернет ничего полезного.) Но если вы попробуете запустить пример, вы обнаружите, что он тут же начинает возвращать значения из ряда Фибоначчи и будет продолжать делать это до тех пор, пока вы продолжаете перебирать его значения. Очевидно, что C# не просто выполняет весь метод целиком перед возвратом.

Чтобы ваш код работал, C# выполняет с ним ряд серьезных операций. Если вы изучите выходные данные компилятора для итератора, например используя инструмент ILDASM (дизассемблер для кода .NET, поставляемый с .NET SDK), вы обнаружите, что он генерирует закрытый вложенный класс, работающий как реализация `IEnumerable<T>`, которую возвращает метод, и одновременно `IEnumerator<T>`, которую возвращает метод `GetEnumerator` интерфейса `IEnumerable<T>`. Код из вашего метода итератора отправляется в метод `MoveNext` этого класса, но его трудно распознать, потому что компилятор разделяет его таким образом, что позволяет каждому `yield return` возвращать управление вызывающей стороне, но продолжать выполнение с места остановки в следующий раз, когда вызывается `MoveNext`. Если необходимо, он будет хранить локальные переменные внутри этого сгенерированного класса, чтобы их значения сохранялись при нескольких вызовах `MoveNext`. Возможно, самый простой способ понять, что делает C# при компиляции итератора, — написать эквивалентный код вручную. В листинге 5.30 выдается та же последовательность Фибоначчи, что и в листинге 5.29, но без помощи итератора. Это не совсем то, что делает компилятор, но это иллюстрирует некоторые возникающие задачи.

Листинг 5.30. Реализация `IEnumerable<T>` вручную

```
public class FibonacciEnumerable :  
    IEnumerable<BigInteger>, IEnumerator<BigInteger>  
{  
    private BigInteger v1;  
    private BigInteger v2;  
    private bool first = true;  
  
    public BigInteger Current => v1;  
  
    public void Dispose() { }  
  
    object IEnumerator.Current => Current;  
  
    public bool MoveNext()  
    {  
        if (first)  
        {  
            v1 = 1;  
            v2 = 1;  
            first = false;  
        }  
        else
```

```
    {
        var tmp = v2;
        v2 = v1 + v2;
        v1 = tmp;
    }

        return true;
    }

    public void Reset()
    {
        first = true;
    }

    public IEnumarator<BigInteger> GetEnumarator() =>
        new FibonacciEnumarable();

    IEnumarator IEnumarable.GetEnumarator() => GetEnumarator();
}
```

Это не особенно сложный пример, потому что его перечислитель, по сути, находится в одном из двух состояний — либо запускается впервые, и поэтому ему нужно выполнить код, который предшествует циклу, либо находится внутри цикла. Тем не менее этот код гораздо сложнее для чтения, чем код из листинга 5.29, потому что механизм поддержки перечисления скрыл основную логику.

Код получился бы еще более запутанным, если бы нам пришлось иметь дело с исключениями. Как я покажу в главах 7 и 8, вы можете использовать блоки `using` и `final`, которые позволяют вашему коду вести себя корректно в случае ошибок, и компилятор может в итоге проделать большую работу для сохранения правильной семантики, когда выполнение метода разделено на несколько итераций. Вам потребуется написать вручную не слишком много перечислений, прежде чем почувствовать благодарность за то, что C# способен делать это за вас⁴.

Кстати, вам не обязательно возвращать `IEnumarable<T>`. Если хотите, то вместо него можете вернуть `IEnumarator<T>`. И, как вы уже видели ранее, объекты,

⁴ Часть этой работы по очистке происходит в вызове `Dispose`. Вспомните, что все реализации `IEnumarator<T>` реализуют `IDisposable`. Ключевое слово `foreach` вызывает `Dispose` после прохода по коллекции (даже если проход был прерван из-за ошибки). Если вы не используете `foreach` и выполняете проход вручную, жизненно важно не забывать вызывать `Dispose`.

которые реализуют любой из этих интерфейсов, также всегда реализуют их необобщенные эквиваленты. Поэтому, если вам нужен простой `IEnumerable` или `IEnumerator`, вам не нужно выполнять дополнительную работу — вы можете передать `IEnumerable<T>` туда, где ожидается простой `IEnumerable`, и это касается также и перечислителей. Если по какой-то причине вы хотите предоставить один из этих нестандартных интерфейсов и не хотите представлять обобщенную версию, вам разрешается писать итераторы, которые возвращают необобщенные формы напрямую.

В случае итераторов следует знать, что они выполняют очень мало кода до тех пор, пока вызывающая сторона не вызовет `MoveNext`. Таким образом, если бы вы пошагово проходили код, который вызывает метод `Fibonacci` в листинге 5.29, то вам бы показалось, что вызов этого метода вообще ничего не делает. Если вы попытаетесь войти в метод в момент, когда он вызывается, никакой код метода не выполнится. Только когда начнется итерация, вы увидите, что тело вашего итератора выполняется. У этого имеется пара последствий.

Первое, что нужно иметь в виду, это то, что, если ваш метод итератора принимает аргументы и вы хотите проверить эти аргументы, вам может потребоваться выполнить дополнительную работу. По умолчанию проверки не произойдет, пока не начнется итерация, поэтому ошибки появятся позже, чем вы могли бы ожидать. Если вы хотите немедленно проверить аргументы, вам потребуется написать оболочку. Листинг 5.31 показывает, как это делается. Он показывает нормальный метод с именем `Fibonacci`, который не использует `yield return` и поэтому не будет особым образом обрабатываться компилятором, как итераторы. Этот нормальный метод проверяет свой аргумент перед вызовом вложенного метода итератора. (Это также показывает, что локальные методы могут использовать `yield return`.)

Листинг 5.31. Проверка аргумента итератора

```
public static IEnumerable<BigInteger> Fibonacci(int count)
{
    if (count < 0)
    {
        throw new ArgumentOutOfRangeException("count");
    }
    return Core(count);

    static IEnumerable<BigInteger> Core(int count)
    {
        BigInteger v1 = 1;
```

```
BigInteger v2 = 2;

for (int i = 0; i < count; ++i)
{
    yield return v1;
    var tmp = v2;
    v2 = v1 + v2;
    v1 = tmp;
}
}
```

Второе, что нужно помнить, это то, что итераторы могут выполняться несколько раз. `IEnumerable<T>` предоставляет метод `GetEnumerator`, который можно вызывать много раз, и тело вашего итератора будет каждый раз запускаться с самого начала. Поэтому хотя ваш метод итератора может быть вызван только один раз, выполнять его может несколько раз.

Collection<T>

Если вы посмотрите на типы в библиотеке классов .NET, то обнаружите, что когда они предлагают свойства, представляющие собой реализацию `IList<T>`, то часто делают это не напрямую. Вместо интерфейса свойства часто предоставляют конкретный тип, хотя обычно это и не `List<T>`. Класс `List<T>` предназначен для использования в качестве части реализации вашего кода, и, если вы предоставляете доступ к нему напрямую, вы, возможно, даете пользователям вашего класса слишком много возможностей. Хотите, чтобы они могли изменять список? И даже если так, разве код не должен узнать, когда это произойдет?

Библиотека классов предоставляет класс `Collection<T>`, предназначенный для использования в качестве базового класса для коллекций, которые тип сделает публичными. Он похож на `List<T>`, но имеет два важных отличия. Во-первых, размер его API меньше — он предлагает `IndexOf`, но все остальные методы поиска и сортировки, доступные для `List<T>`, отсутствуют, и нет способа выяснить или изменить его емкость независимо от его размера. Во-вторых, он позволяет производным классам определять, когда элементы были добавлены или удалены. `List<T>` так не делает на том основании, что это ваш список, поэтому вы, вероятно, знаете о том, что добавляете и удаляете элементы. Механизмы уведомлений тоже используют ресурсы, поэтому `List<T>` избегает ненужных затрат, просто не предлагая этих мето-

дов. Но `Collection<T>` предполагает, что внешний код будет иметь доступ к вашей коллекции, а вы не сможете контролировать каждое добавление и удаление, что оправдывает расходы, связанные с предоставлением вам информации об изменении списка. (Это доступно только для кода, производного от `Collection<T>`. Если вы хотите, чтобы код, использующий вашу коллекцию, мог обнаруживать изменения, то для этого предназначен тип `ObservableCollection<T>`.)

Как правило, вы наследуете класс от `Collection<T>` и имеете возможность переопределить его виртуальные методы, чтобы отслеживать изменения коллекции. (О наследовании и переопределении — в главе 6.) `Collection<T>` реализует как `IList`, так и `IList<T>`, так что вы можете передавать основанную на `Collection<T>` коллекцию через свойство типа интерфейса, но обычно производный тип делают открытым и используют его вместо интерфейса в качестве типа свойства.

ReadOnlyCollection<T>

Если требуется немодифицируемая коллекция, то вместо `Collection<T>` можно рассмотреть использование `ReadOnlyCollection<T>`. Этот интерфейс накладывает еще больше ограничений, чем массивы: вы не только не можете добавлять, удалять или вставлять элементы, нельзя их даже заменять. Этот класс реализует `IList<T>`, который требует индексатора с `get`, и `set`, но `set` при этом выдает исключение. (Конечно, он также реализует и `IReadOnlyCollection<T>`.)

Если тип элемента вашей коллекции является ссылочным, то создание коллекции только для чтения не препятствует изменению объектов, на которые ссылаются элементы. Я могу получить, скажем, двенадцатый элемент из коллекции, доступной только для чтения, и он вернет мне ссылку. Получение ссылки считается операцией только для чтения, но теперь, когда я получил эту ссылку, объект коллекции больше не участвует в процессе, и я могу делать с этой ссылкой все, что мне хочется. Поскольку C# не содержит концепции ссылки только для чтения (здесь нет ничего похожего на константные ссылки C++), единственный способ представить коллекцию, доступную только для чтения, — это использовать неизменяемый тип в сочетании с `ReadOnlyCollection<T>`.

Есть два пути использования `ReadOnlyCollection<T>`. Вы можете задействовать его непосредственно как обертку для существующего списка — его

конструктор использует `IList<T>`, и это обеспечит доступ только для чтения. (`List<T>`, кстати, содержит метод `AsReadOnly`, который создает для вас обертку только для чтения.) В качестве альтернативы вы можете наследовать от него класс. Как и с `Collection<T>`, некоторые классы проделывают это для коллекций, которые они хотят открыть через свойства. Обычно это происходит потому, что им требуется определить дополнительные методы, обусловленные задачами коллекции. Но даже если вы наследуете от этого класса, вы все равно будете использовать его в качестве обертки для лежащего в основе списка, поскольку единственный конструктор, который он предоставляет, — это тот, который принимает список.



`ReadOnlyCollection<T>` обычно не подходит для сценариев, которые автоматически сопоставляют объектные модели и внешнее представление. Например, это вызывает проблемы в типах, используемых в качестве объектов передачи данных (DTO), которые преобразуются в сообщения и из сообщений JSON, отправляемых по Сети, а также в системах объектно-реляционного отображения, которые представляют содержимое базы данных через объектную модель. Платформы для этих сценариев должны иметь возможность создавать экземпляры ваших типов и заполнять их данными, поэтому, хотя коллекция только для чтения может концептуально соответствовать некоторой части вашей модели, она не способна инициализировать объекты так, как того ожидают указанные библиотеки отображения.

Обращение к элементам по индексу и синтаксис диапазона

Будь то использование массивов, `List<T>`, `IList<T>` или других родственных типов и интерфейсов, мы обращались к элементам простыми способами, такими как `items[0]`, а если говорить в более общем смысле, с помощью выражений вида `arrayOrListExpression[indexExpression]`. До сих пор во всех примерах для индекса использовалось выражение типа `int`, но это далеко не единственный вариант. Листинг 5.32 обращается к последнему элементу массива, используя альтернативный синтаксис.

Листинг 5.32. Доступ к последнему элементу массива через индекс с отсчетом от конца

```
char[] letters = { 'a', 'b', 'c', 'd' };
char lastLetter = letters[^1];
```

Пример демонстрирует один из двух новых операторов, введенных в C# 8.0 для использования в индексаторах: оператор ^ и оператор диапазона. Последний показан в листинге 5.33 и представляет собой пару точек (...). Он используется для определения поддиапазонов массивов, строк или любого индексируемого типа, который реализует определенный шаблон.

Листинг 5.33. Получение поддиапазона массива с помощью оператора диапазона

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };
// Получает 4-й и 5-й элементы (но не 3-й по причинам, о которых ниже)
int[] theFourthTheFifth = numbers[3..5];
```

Выражения с использованием операторов ^ и .. имеют тип `Index` и `Range` соответственно. Эти типы встроены в .NET Core 3.0 и .NET Standard 2.1. На момент написания книги Microsoft еще не выпустила пакет NuGet, определяющий версии этих типов для более старых целевых платформ, а это означает, что эти новые языковые функции доступны только в самых новых средах выполнения. Вполне возможно, что к тому времени, когда вы будете это читать, уже появится пакет NuGet, который сделает `Index` и `Range` более доступными, поскольку большая часть их работы не зависит от новых возможностей среды выполнения. Однако теперь, согласно политике Microsoft, новые функции языка C# привязаны к новым версиям .NET, поэтому нам еще предстоит выяснить уровень поддержки новых пакетов в старых версиях среды выполнения.

System.Index

Вы можете поместить оператор ^ перед любым выражением `int`. Он создает `System.Index`, значимый тип, представляющий собой позицию. Когда вы создаете индекс с помощью ^, он считается относительно конца, но вы также можете создавать индексы относительно начала. Для этого нет специального оператора, но, поскольку `Index` предлагает неявное преобразование из `int`, вы можете просто присвоить значения `int` переменным типа `Index` непосредственно, как показано в листинге 5.34. Также индекс можно создать явно, как это показано в строке с `var`. Последний аргумент `bool` является необязательным, и по умолчанию он равен `false`, но с его помощью я показываю, как `Index` узнает, какой именно вариант вам требуется.

Как показано в листинге 5.34, индексы с отсчетом от конца существуют независимо от какой-либо конкретной коллекции. (Внутри `Index` хранит

индексы с отсчетом от конца в виде отрицательных чисел. Это означает, что индекс имеет тот же размер, что и `int`. Это также означает, что отрицательные индексы с отсчетом от конца недопустимы — вы получите исключение, если попытаетесь их создать.) При использовании индекса C# генерирует код, который определяет фактическую позицию элемента. Если `small` и `big` являются массивами с 3 и 30 элементами соответственно, `small[last]` вернет третий, а `big[last]` вернет тридцатый элемент. C# превратит эти обращения в `small[last.GetOffset.Length)` и `big[last.GetOffset.Length)` соответственно.

Листинг 5.34. Индексы с отсчетом от начала и от конца

```
Index first = 0;
Index second = 1;
Index third = 2;
var fourth = new Index(3, fromEnd: false);

Index antePenultimate = ^3;
Index penultimate = ^2;
Index last = ^1;
Index directlyAfterTheLast = ^0;
```

Часто говорят, что три самые сложные проблемы в вычислительной технике — это выбор имен и ошибки смещения на единицу. На первый взгляд листинг 5.34 показывает, что `Index` только способствует этим проблемам. Кого-то может раздражать, что индекс третьего элемента равен двум, а не трем, но это, по крайней мере, согласуется с тем, как массивы работали в C# изначально, и является нормой для любой системы индексации от нуля. Но если учесть это соглашение касательно нуля, то с какой стати индексы с отсчетом от конца начинаются с единицы? Мы обозначаем первый элемент с помощью `0`, а последний — с помощью `^1`!

На это есть веские причины. Основная идея заключается в том, что в C# индексы всегда указывали расстояние. Когда разработчики языка программирования выбирают систему индексации с нулем, на самом деле это не решение назвать первый элемент `0`: это решение интерпретировать индекс как расстояние от начала массива. Результатом этого является то, что индекс на самом деле ссылается не на элемент. Рисунок 5.1 показывает коллекцию из четырех элементов и указывает, куда в этой коллекции будут указывать различные значения индекса. Обратите внимание, что все индексы относятся к границам между элементами. Это может показаться

избыточной тонкостью, но это ключ к пониманию всех систем индексов с отсчетом от нуля, и это же лежит в основе видимой несогласованности в листинге 5.34.

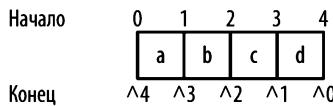


Рис. 5.1. Куда указывает индекс

Когда вы обращаетесь к элементу коллекции по индексу, то запрашиваете элемент, который начинается с позиции, указанной индексом. Поэтому `array[0]` извлекает единственный элемент, который начинается в начале массива, т. е. тот, который заполняет пространство между индексами 0 и 1. Аналогично `array[1]` извлекает элемент между индексами 1 и 2. Что будет означать `array[^0]`? Это было бы попыткой получить элемент, который начинается в самом конце массива⁵. Поскольку все элементы занимают определенное пространство, элемент, который начинается в самом конце массива, обязательно заканчивается на одну позицию дальше. В этом примере четырехэлементного массива обращение `array[^0]` эквивалентно `array[4]`. Следовательно, мы запрашиваем элемент, занимающий пространство, которое начинается через четыре элемента с начала и заканчивается через пять элементов. А поскольку массив состоит из четырех элементов, обращение не будет работать. Очевидное расхождение — `array[0]` запрашивает первый элемент, а нужно написать `array[^1]`, чтобы получить последний, — происходит потому, что элементы расположены между двумя индексами и индексаторы массива всегда извлекают элемент между указанным индексом и индексом после него. Тот факт, что они делают это, даже когда вы указали индекс с отсчетом от конца, и является причиной того, что они как будто начинают отсчет от единицы. Эту языковую функцию можно было разработать иначе: вполне можно представить правило, при котором индексы с отсчетом от конца всегда обращаются к элементу,

⁵ Поскольку индексы с отсчетом от конца хранятся в виде отрицательных чисел, вы можете задаться вопросом, является ли `^0` вообще допустимым, поскольку тип `int` не различает положительный и отрицательный ноль. Он допустим, потому что, как вы скоро увидите, `^0` полезен при использовании диапазонов. Чтобы это работало, `Index` корректирует конечные относительные индексы на единицу, а также делает их отрицательными, поэтому он хранит `^0` как `-1`, `^1` — как `-2` и т. д.

который заканчивается на указанном расстоянии от конца, а начинается на одну позицию раньше. В этом была бы приятная симметрия, потому что тогда обращение `array[^0]` ссылалось бы на последний элемент. Но это бы вызвало больше проблем, чем решило.

Было бы странно, если бы индексаторы интерпретировали индекс двумя разными способами — это означало бы, что два разных индекса могут ссылаться на одну и ту же позицию и одновременно извлекать разные элементы. В любом случае разработчики на C# уже привыкли к текущему положению дел. Как показано в листинге 5.35, до C# 8.0 для доступа к последнему элементу массива использовался индекс, получаемый путем вычитания единицы из длины. И если вы хотите получить предпоследний элемент, вам нужно вычесть два из длины и т. д. Как видите, новый синтаксис с отсчетом от конца полностью согласуется с давно существующей практикой.

Листинг 5.35. Индексы с отсчетом от конца и их эквиваленты до появления типа `Index`

```
int lastOld = numbers[numbers.Length - 1];
int lastNew = numbers[^1];

int penultimateOld = numbers[numbers.Length - 2];
int penultimateNew = numbers[^2];
```

О чём еще стоит задуматься, так это о том, как бы все выглядело, если бы мы обращались к массивам с указанием диапазонов. Первый элемент находится в диапазоне 0–1, а последний — в диапазоне ^1–^0. В этих выражениях прослеживается четкая симметрия между формой с отсчетом от начала и с отсчетом от конца. И раз уж мы заговорили о диапазонах...

System.Range

Как я уже говорил ранее, в C# 8.0 добавлены два новых оператора для работы с массивами и другими индексируемыми типами. Мы только что рассмотрели `^` и соответствующий ему тип `Index`. Другой называется *оператором диапазона* и имеет соответствующий тип `Range`, также расположенный в пространстве имен `System`. Диапазон — это просто пара значений индекса, которые доступны через свойства `Start` и `End`. Конструктор `Range` принимает два значения `Index`, но в C# характерным способом его создания будет использование оператора `range`, как показано в листинге 5.36.

Листинг 5.36. Различные диапазоны

```
Range everything = 0..^0;
Range alsoEverything = 0..;
Range everythingAgain = ..^0;
Range everythingOneMoreTime = ..;
var yetAnotherWayToSayEverything = Range.All;

Range firstThreeItems = 0..3;
Range alsoFirstThreeItems = ..3;

Range allButTheFirstThree = 3..^0;
Range alsoAllButTheFirstThree = 3..;

Range allButTheLastThree = 0..^3;
Range alsoAllButTheLastThree = ..^3;

Range lastThreeItems = ^3..^0;
Range alsoLastThreeItems = ^3..;
```

Как видите, если не указывать начальный индекс перед `..`, он по умолчанию равен `0`, а если опустить конец, то он по умолчанию станет равным `^0` (т. е. самому началу и концу соответственно). В примере также показано, что начало может быть как с отсчетом от начала, так и от конца, как и, собственно, конец.



Значение по умолчанию для `Range` — то, которое вы получите в поле или элементе массива, который вы не инициализируете явным, т. е. `0..0`. Это обозначение пустого диапазона. Хотя это естественный результат того, что значимые типы по умолчанию всегда инициализируются в нулевые значения, это может оказаться не тем, что вы ожидаете, учитывая, что `..` эквивалентно `Range.All`.

Поскольку `Range` работает в терминах `Index`, начало и конец обозначают смещения, а не элементы. Например, рассмотрим, что будет означать диапазон `1..3` для элементов, показанных на рис. 5.1. В этом случае оба индекса отсчитываются от начала. Начальный индекс `1` является границей между первым и вторым элементами (`a` и `b`), а конечный индекс `3` — границей между третьим и четвертым элементами (`c` и `d`). Так что в целом это диапазон, который начинается в начале `b` и заканчивается в конце `c`, как показано на рис. 5.2. Таким образом, он определяет двухэлементный диапазон (`b` и `c`).

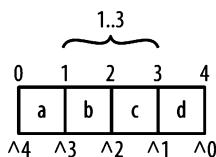


Рис. 5.2. Диапазон

Интерпретация диапазонов на первый взгляд удивляет: некоторые ожидают, что `1..3` будет указывать на первый, второй и третий элементы (или, если они принимают во внимание индексацию C# с отсчетом от нуля, возможно, второй, третий и четвертый элементы). Поначалу может показаться непоследовательным, что начальный индекс включительный, а конечный — исключительный. Но как только вы вспомните, что индекс относится не к элементу, а к смещению и, следовательно, к границе между двумя элементами, все сразу встает на свои места. Если вы нарисуете позиции, представленные индексами диапазона, как показано на рис. 5.2, станет совершенно очевидно, что диапазон, обозначенный `1..3`, охватывает только два элемента.

Итак, что же мы можем делать с `Range`? Листинг 5.33 показывает, что мы можем использовать его для получения поддиапазона массива. Это создаст новый массив соответствующего размера и скопирует в него значения из диапазона. Этот же синтаксис сработает и для получения подстрок, как показано в листинге 5.37.

Листинг 5.37. Получение подстроки с помощью диапазона

```
string t1 = "dysfunctional";
string t2 = t1[3..6];
Console.WriteLine($"Putting the {t2} in {t1}");
```

Еще вы можете использовать `Range` совместно с `ArraySegment<T>`, значимым типом, который ссылается на диапазон элементов в массиве. Листинг 5.38 вносит небольшое изменение в листинг 5.33. Вместо того чтобы передавать индексатору массива диапазон, сначала он создает `ArraySegment<int>`, который представляет весь массив, а затем использует диапазон, чтобы получить второй `ArraySegment<int>`, представляющий четвертый и пятый элементы. Преимущество здесь в том, что не нужно выделять новый массив — значения обоих `ArraySegment<int>` ссылаются на один и тот же базовый массив, указывая на разные его части, а поскольку `ArraySegment<int>` — это значимый тип, мы можем избежать выделения новых блоков в куче. (Массив `Segment<int>`,

кстати, не поддерживает диапазоны напрямую. Компилятор превращает это в вызов метода сегмента `Slice`.)

Листинг 5.38. Получение поддиапазона `ArraySegment<T>` с помощью оператора диапазона

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };
ArraySegment<int> wholeArrayAsSegment = numbers;
ArraySegment<int> theFourthTheFifth = wholeArrayAsSegment[3..5];
```

Тип `ArraySegment<T>` существует со временем .NET 2.0 (и был в .NET Standard с версии 1.0). Он позволяет избежать дополнительных выделений, но имеет ограничение: он работает только с массивами. А что насчет строк? .NET Core 2.1 добавил более общую форму этой концепции, `Span<T>` (она также доступна для более старых версий .NET благодаря NuGet-пакету `System.Memory`). Подобно `ArraySegment<T>`, `Span<T>` представляет подпоследовательность элементов внутри чего-то еще, но он гораздо более гибок в отношении того, чем может быть это самое «что-то еще». Это может быть массив, но также может быть и строка, память в стековом фрейме или память, выделенная какой-либо библиотекой или системным вызовом, полностью находящимися вне .NET. Тип `Span<T>` более подробно обсуждается в главе 18, а сейчас с помощью листинга 5.39 я проиллюстрирую его использование (и его аналог только для чтения).

Листинг 5.39. Получение поддиапазона `Span<T>` с помощью оператора диапазона

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };
Span<int> wholeArrayAsSpan = numbers;
Span<int> theFourthTheFifth = wholeArrayAsSpan[3..5];
ReadOnlySpan<char> textSpan = "dysfunctional".AsSpan();
ReadOnlySpan<char> such = textSpan[3..6];
```

Их логика такая же, как и у предыдущих примеров, но они избегают создания копий исходных данных.

Итак, мы увидели, что можем использовать диапазоны нескольких типов: массивы, строки, `ArraySegment<T>` и `Span<T>`. Возникает вопрос: имеется ли в C# список типов, которые обрабатываются особым образом, или мы можем использовать индексаторы и диапазоны в наших собственных типах? Ответы соответственно да и да. В C# внедрена кое-какая обработка массивов и строк: он вызывает определенные методы библиотеки классов .NET для создания подмассивов и подстрок. Тем не менее особой обработки сегментов массива или диапазонов нет: они работают, потому что соответствуют шаблону. Существует также шаблон, позволяющий использовать `Index`. Если

вы поддерживаете такие же шаблоны, вы можете заставить `Index` и `Range` работать со своими типами.

Поддержка `Index` и `Range` в ваших собственных типах

Тип массива не определяет индексатор, который принимает аргумент типа `Index`. Не делает этого и никакой из обобщенных подобных массивам типов, показанных ранее в этой главе. Все они содержат обычные индексаторы, основанные на `int`, но тем не менее вы можете использовать с ними и `Index`. Как я объяснял ранее, код вида `col[index]` будет расширен до `col[index.GetOffset(a.Length)]`. Поэтому все, что вам нужно, — это индексатор на основе `int` и свойство типа `int` с именем `Length` или `Count`⁶. Листинг 5.40 показывает минимальный объем работы, который необходимо выполнить, чтобы код мог передавать `Index` индексатору вашего типа. Это не очень полезная реализация, но ее достаточно, чтобы удовлетворить C#.

Листинг 5.40. `Index` минимальными усилиями

```
public class Indexable
{
    public char this[int index] => (char)('0' + index);

    public int Length => 10;
}
```

`IList<T>` соответствует требованиям шаблона (как и типы, которые его реализуют, такие как `List<T>`), так что вы можете передать `Index` в индексатор всего, что его реализует. (Он предоставляет свойство `Count` вместо `Length`, но шаблон принимает и то и другое.) Реализация этого интерфейса встречается очень часто, поэтому на практике многие типы, которые были определены задолго до C# 8.0, автоматически получают поддержку `Index` без необходимости вносить изменения. Это пример того, как основанная на шаблонах поддержка `Index` позволяет библиотекам, предназначенным для более старых версий .NET (например, .NET Standard 2.0), где `Index` недоступен, определять типы, которые будут работать с `Index` при использовании .NET Core 3.0. или позже.

⁶ В тех случаях, когда вы используете `^` непосредственно с `int` внутри индексатора массива (например, `[^i]`, где `i` имеет тип `int`), компилятор генерирует несколько более простой код. Вместо преобразования `i` в `Index` и последующего вызова `GetOffset` он генерирует код, эквивалентный `a[a.Length - i]`.



Есть способ еще проще: определите индексатор, который принимает аргумент типа `Index`. Однако большинство индексируемых типов представляют индексатор на основе `int`, поэтому на практике вам придется перегружать индексатор, чтобы он предлагал обе формы. Это не проще, но позволит вашему коду различать индекссы с отсчетом от начала и от конца. Если мы используем `1` или `^9` с листингом 5.40, его индексатор в обоих случаях увидит `1`, потому что C# генерирует код, который преобразует `Index` в `int` с отсчетом от начала, но, если вы напишите индексатор с параметром `Index`, C# будет передавать именно `Index`. Если вы перегружаете индексатор, делая доступными обе формы, `int` и `Index`, он не будет генерировать код, который использует индекс типа `int`: шаблон включается только в том случае, если недоступен индексатор, использующий `Index`.

Шаблон для поддержки `Range` отличается: если ваш тип содержит метод экземпляра `Slice`, который принимает два целочисленных аргумента, C# позволит коду передавать `Range` в качестве аргумента индексатора. Листинг 5.40 показывает минимальный объем работы, чтобы включить эту поддержку для своего типа, хотя это не самая полезная реализация. (Как и в случае с `Index`, в качестве альтернативы вы можете просто определить перегрузку индексатора, которая принимает `Range` напрямую. Но опять же, преимущество шаблонного подхода состоит в том, что вы можете использовать его при ориентации на более старые версии, например на .NET Standard 2.0, которые не содержат типов `Range` или `Index`, хотя поддерживают диапазоны для кода, предназначенного для более новых версий.)

Листинг 5.41. Range минимальными усилиями

```
public class Rangeable
{
    public int Length => 10;

    public Rangeable Slice(int offset, int length) => this;
}
```

Вы могли заметить, что показанный тип не определяет индексатор. Это потому, что для основанной на шаблонах поддержки выражений вида `x[1..^1]` это не нужно. На первый взгляд похоже, что мы используем индексатор, но на самом деле здесь просто вызывается метод `Slice`. (Аналогично более ранние примеры диапазонов со `string` и массивами компилируются в вызовы методов.) Вам необходимо иметь свойство `Length` (или `Count`), потому

что компилятор генерирует код, который рассчитывает на них, чтобы иметь возможность разрешить индексы диапазона. Листинг 5.42 показывает, как компилятор использует типы, поддерживающие этот шаблон.

Листинг 5.42. Как расширяется индексирование диапазона

```
Rangeable r1 = new Rangeable();
Range r = 2..^2;

Rangeable r2;

r2 = r1[r];
// эквивалентно
int startIndex = r.Start.GetOffset(r1.Length);
int endIndex = r.End.GetOffset(r1.Length);
r2 = r1.Slice(startIndex, endIndex - startIndex);
```

До сих пор все рассмотренные коллекции были линейными: я показал только простые последовательности объектов, некоторые из которых предлагают индексированный доступ. Однако .NET предоставляет и другие виды коллекций.

Словари

Одним из наиболее полезных типов коллекций является словарь. .NET определяет класс `Dictionary< TKey, TValue >`, а также соответствующий интерфейс с ожидаемым именем `IDictionary< TKey, TValue >` и версией только для чтения, `IReadOnlyDictionary < TKey, TValue >`. Они представляют собой коллекции пар `ключ/значение`, и их наиболее полезная возможность состоит в поиске значения на основе ключа, что делает словари полезными для представления ассоциаций.

Предположим, вы пишете пользовательский интерфейс для приложения, которое поддерживает онлайн-обсуждения. При отображении сообщения вам может потребоваться показать сведения о пользователе, который его отправил, например его имя и изображение, и вам вряд ли захочется искать эти сведения каждый раз, где бы они ни хранились. Если пользователь общается с несколькими друзьями, одни и те же люди будут встречаться неоднократно, поэтому вы задумаетесь об использовании какого-то кэша, чтобы избежать повторного поиска. В качестве части такого кэша вы можете использовать словарь. Листинг 5.43 схематично демонстрирует этот подход.

(В нем пропущены специфичные для приложения детали, касающиеся, например, того, каким именно образом данные получаются и когда старые данные удаляются из памяти.)

Листинг 5.43. Использование словаря в качестве части кэша

```
public class UserCache
{
    private readonly Dictionary<string, UserInfo> _cachedUserInfo =
        new Dictionary<string, UserInfo>();

    public UserInfo GetInfo(string userHandle)
    {
        RemoveStaleCacheEntries();
        if (!_cachedUserInfo.TryGetValue(userHandle, out UserInfo info))
        {
            info = FetchUserInfo(userHandle);
            _cachedUserInfo.Add(userHandle, info);
        }
        return info;
    }

    private UserInfo FetchUserInfo(string userHandle)
    {
        // получить информацию ...
    }

    private void RemoveStaleCacheEntries()
    {
        // логика конкретного приложения, решающая, когда удалять старые
        // записи ...
    }
}

public class UserInfo
{
    // пользовательская информация, зависящая от приложения ...
}
```

Первый аргумент типа, `TKey`, используется для поиска, и в этом примере я использую строку, которая каким-то образом идентифицирует пользователя. Аргумент `TValue` — это значимый тип, связанный с ключом, — в данном случае информация, ранее извлеченная для пользователя и кэшированная локально в экземпляре `UserInfo`. Метод `GetInfo` использует `TryGetValue` для поиска в словаре данных, связанных с дескриптором пользователя. Суще-

ствует и более простой способ получения значения. Как показано в листинге 5.44, словари предоставляют индексатор. Однако он выдает исключение `KeyNotFoundException`, если нет записи с указанным ключом. Это хорошо, когда ваш код всегда ожидает найти то, что ищет, но в нашем случае ключ будет отсутствовать для любого пользователя, данные о котором еще не занесены в кэш. Скорее всего, это будет происходить довольно часто, поэтому я использую `TryGetValue`. В качестве альтернативы можно использовать метод `ContainsKey` для того, чтобы перед извлечением увидеть, существует ли запись, но это неэффективно, если значение все же есть. В таком случае словарь будет искать ее дважды: один раз при вызове `ContainsKey`, а затем при использовании индексатора. `TryGetValue` выполняет тест и поиск как одну операцию.

Листинг 5.44. Поиск по словарю с индексатором

```
UserInfo info = _cachedUserInfo[userHandle];
```

Как и следовало ожидать, мы также можем использовать индексатор для установки значения, связанного с ключом. В листинге 5.43 я этого не делал. Взамен я использовал метод `Add`, потому что у него немного другая семантика: вызывая `Add`, вы сигнализируете, что, по вашему мнению, записи с указанным ключом еще не существует. В то время как индексатор словаря без возражений перезапишет существующую запись, `Add` выдаст исключение, если вы попытаетесь использовать ключ, для которого запись уже существует. В ситуациях, когда использование уже существующего ключа означает, что что-то пошло не так, лучше вызвать `Add`, чтобы проблема не осталась незамеченной.

Интерфейс `IDictionary< TKey, TValue >` требует от своих реализаций также предоставить интерфейс `ICollection<KeyValuePair< TKey, TValue >>`, а значит, и `IEnumerable<KeyValuePair< TKey, TValue >>`. Вариант только для чтения требует последнего, но не первого. Эти интерфейсы зависят от обобщенной структуры `KeyValuePair< TKey, TValue >`, которая является простым контейнером, который сохраняет ключ и значение в одном экземпляре. Это означает, что вы можете перебирать словарь, используя `foreach`, и он будет по очереди возвращать каждую пару ключ/значение.

Наличие `IEnumerable< T >` и метода `Add` также означает, что мы можем использовать синтаксис инициализатора коллекции. Это не совсем то же самое, что мы видели в случае с простым списком, потому что `Add` словаря принимает два аргумента: ключ и значение. Однако синтаксис инициализатора

коллекции может работать с многоаргументными методами `Add`. Каждый набор аргументов нужно заключить во вложенные фигурные скобки, как показано в листинге 5.45.

Листинг 5.45. Синтаксис инициализатора коллекции в случае словаря

```
var textToNumber = new Dictionary<string, int>
{
    { "One", 1 },
    { "Two", 2 },
    { "Three", 3 },
};
```

Как вы видели в главе 3, существует альтернативный способ заполнения словаря: вместо использования инициализатора коллекции вы можете использовать синтаксис инициализатора объекта. Как вы, возможно, помните, он позволяет вам устанавливать свойства для вновь созданного объекта. Это единственный способ инициализировать свойства анонимного типа, но вы можете использовать его для любого типа. Индексаторы — это просто особый вид свойств, поэтому имеет смысл устанавливать их с помощью инициализатора объекта. Хотя в главе 3 об этом уже говорилось, есть смысл сравнить инициализаторы объектов с инициализаторами коллекций, поэтому в листинге 5.46 показан альтернативный способ инициализации словаря.

Листинг 5.46. Синтаксис инициализатора объекта в случае словаря

```
var textToNumber = new Dictionary<string, int>
{
    ["One"] = 1,
    ["Two"] = 2,
    ["Three"] = 3
};
```

Хотя результат здесь будет таким же, как и в листингах 5.45 и 5.46, для каждого из них компилятор генерирует немного другой код. В листинге 5.45 он заполняет коллекцию путем вызова `Add`, тогда как в листинге 5.46 используется индексатор. Для `Dictionary< TKey, TValue >` результат будет таким же, поэтому нет объективной причины выбирать один или другой, но для некоторых классов разница может иметь значение. Например, если вы используете класс, у которого есть индексатор, но нет метода `Add`, будет работать только код на основе индекса. Кроме того, с помощью синтаксиса инициализатора объекта можно было бы установить как индексированные

значения, так и свойства для типов, которые это поддерживают (хотя вы не сможете проделать это с `Dictionary< TKey, TValue >`, потому что у него нет доступных для записи свойств, кроме индексатора).

Для обеспечения быстрого поиска класс коллекции `Dictionary< TKey, TValue >` полагается на хеши. В главе 3 описан метод `GetHashCode`, и вы должны убедиться, что любой тип, который вы используете в качестве ключа, обеспечивает хорошую реализацию хеша. Класс `string` отлично подойдет. Метод `GetHashCode` по умолчанию является надежным только в том случае, если разные экземпляры типа всегда имеют разные значения, но из типов, для которых это так, получаются отличные ключи. Кроме того, класс словаря предоставляет конструкторы, которые принимают `IEqualityComparer< TKey >`, что позволяет реализовать `GetHashCode` и `Equals` для использования вместо тех, которые предоставляются самим типом ключа. Листинг 5.47 использует это для создания не зависящей от регистра версии листинга 5.45.

Листинг 5.47. Словарь без учета регистра

```
var textToNumber =
    new Dictionary<string, int>(StringComparer.InvariantCultureIgnoreCase)
{
    { "One", 1 },
    { "Two", 2 },
    { "Three", 3 },
};
```

При этом используется класс `StringComparer`, который предоставляет различные реализации `IComparer< string >` и `IEqualityComparer< string >`, предлагающие разные правила сравнения.

В данном случае я выбрал порядок, который игнорирует регистр и настройки локали, обеспечивая одинаковое поведение в разных регионах. Если бы я отображал строки, то, вероятно, использовал одно из сравнений с учетом культурных особенностей.

Сортированные словари

Из-за того что `Dictionary< TKey, TValue >` использует поиск на основе хеша, порядок, в котором он возвращает элементы при переборе его содержимого, труднопредсказуем и не очень удобен. Как правило, он не будет иметь никакого отношения к порядку, в котором добавлялось содержимое, и не

будет иметь очевидной связи с самим содержимым. (Порядок, как правило, выглядит случайным, хотя на самом деле он основан на хеш-кодах.)

Иногда бывает полезно иметь возможность извлекать содержимое словаря в каком-то более осмысленном порядке. Вы всегда можете поместить содержимое в массив и затем отсортировать его, но пространство имен `System.Collections.Generic` содержит еще две реализации интерфейса `IDictionary< TKey, TValue >`, которые постоянно хранят свое содержимое в определенном порядке. Имеется `SortedDictionary< TKey, TValue >` и менее очевидно озаглавленный `SortedList< TKey, TValue >`, который, несмотря на название, реализует интерфейс `IDictionary< TKey, TValue >`, но не реализует напрямую `IList< T >`.

Эти классы не используют хеш-коды. Они обеспечивают по-прежнему достаточно быстрый поиск за счет сортировки содержимого. Они стараются сохранить порядок каждый раз, когда вы добавляете новую запись, что делает добавление в обоих классах довольно медленным по сравнению со словарем на основе хеша, но это означает и то, что, когда вы перебираете содержимое, данные выдаются по порядку. Как и в случае сортировки массивов и списков, вы можете указать собственную логику сравнения, но если вы ее не предоставите, то этим словарям для реализации `IComparable< T >` потребуется тип ключа.

Сортировка `SortedDictionary< TKey, TValue >` очевидна только при использовании поддержки перечисления (например, через `foreach`). `SortedList< TKey, TValue >` также перечисляет свое содержимое по порядку, но дополнительно предоставляет доступ к ключам и значениям посредством цифровой индексации. Это работает не через индексатор объекта, который, как и любой словарь, ожидает передачи ключа. Вместо этого словарь в виде отсортированного списка определяет два свойства, `Keys` и `Values`, которые предоставляют все ключи и значения в виде `IList< TKey >` и `IList< TValue >` соответственно, отсортированные так, чтобы ключи располагались в порядке возрастания.

Вставка и удаление объектов является относительно затратной операцией для отсортированного списка, поскольку должна сдвигать содержимое списка ключей и значений вверх или вниз. (Это означает, что одиночная вставка имеет сложность $O(n)$.) С другой стороны, упорядоченный словарь использует древовидную структуру данных для сортировки своего содержимого.

Точные детали неизвестны, но производительность вставки и удаления имеет задокументированную сложность $O(\log n)$, что намного лучше, чем

у упорядоченного списка. Однако такая более сложная структура данных вынуждает упорядоченный словарь занимать значительно больший объем памяти⁷. Это означает, что ни один из них не будет явно быстрее или лучше другого — все зависит от модели использования, поэтому .NET и предоставляет оба варианта.

В большинстве случаев `Dictionary< TKey, TValue >` на основе хеша обеспечивает лучшую производительность вставки, удаления и поиска, чем любой из упорядоченных словарей, а также более низкое потребление памяти, чем `SortedDictionary< TKey, TValue >`. Поэтому вы должны использовать эти отсортированные словарные коллекции только в случае, если вам нужен доступ к содержимому словаря по порядку.

Множества

В пространстве имен `System.Collections.Generic` определен интерфейс `ISet< T >`. Его модель работы проста: конкретное значение либо входит в множество, либо нет. Вы можете добавлять или удалять элементы, но множество не отслеживает, сколько раз вы добавили элемент. Также `ISet< T >` не требует, чтобы элементы хранились в каком-то определенном порядке.

Все типы множеств реализуют `ICollection< T >`, который и предоставляет методы для добавления и удаления элементов. На самом деле он также определяет метод определения вхождения. Хотя я и не обращал на это вашего внимания, в листинге 5.24 вы можете увидеть, что `ICollection< T >` определяет метод `Contains`. Он принимает одно значение и возвращает `true`, если это значение содержится в коллекции.

Учитывая, что `ICollection< T >` уже предоставляет основные операции для множества, вы можете задаться вопросом, зачем нам вообще нужен `ISet< T >`. Тем не менее он добавляет ряд вещей. Хотя `ICollection< T >` определяет метод `Add`, `ISet< T >` определяет свою слегка отличающуюся версию, которая возвращает `bool`. С ее помощью вы можете узнать, был ли только что добавленный элемент во множестве до добавления. Листинг 5.48 использует этот функционал для обнаружения дубликатов в методе, который отображает каждую строку во входных данных по одному разу. (Это хорошая

⁷ Применяются обычные оговорки касательно анализа сложности — для небольших коллекций более простая структура данных вполне может оказаться выигрышной, ее теоретическое преимущество вступает в силу только в случае больших коллекций.

иллюстрация использования, но на практике было бы проще использовать оператор `Distinct LINQ`, описанный в главе 10.)

Листинг 5.48. Использование множества для определения новых элементов

```
public static void ShowEachDistinctString(IEnumerable<string> strings)
{
    var shown = new HashSet<string>(); // Реализуем ISet<T>

    foreach (string s in strings)
    {
        if (shown.Add(s))
        {
            Console.WriteLine(s);
        }
    }
}
```

`ISet<T>` также определяет некоторые операции для объединения множеств. Метод `UnionWith` принимает `IEnumerable<T>` и добавляет ко множеству все значения из этой последовательности, которых в нем еще нет. Метод `ExceptWith` удаляет из множества элементы, которые также находятся в переданной последовательности. Метод `IntersectWith` удаляет из множества элементы, которые не входят в передаваемую последовательность. И наконец, `SymmetricExceptWith` также принимает последовательность и удаляет из множества элементы, которые содержатся в последовательности, но также добавляет ко множеству значения из последовательности, которые ранее в нем отсутствовали.

Есть также ряд методов для сравнения множеств. Опять же, все они принимают аргумент `IEnumerable<T>`, представляющий собой другое множество, с которым необходимо выполнять сравнение. `IsSubsetOf` и `IsProperSubsetOf` позволяют проверить, содержит ли множество, для которого вы вызываете метод, только элементы, которые также присутствуют в последовательности, причем последний метод дополнительно требует, чтобы последовательность содержала хотя бы один элемент, отсутствующий во множестве. `IsSupersetOf` и `IsProperSupersetOf` выполняют такие же проверки в противоположном направлении. Метод `Overlaps` сообщает, имеют ли два множества хотя бы один общий элемент.

Математические наборы не определяют порядок своего содержимого, поэтому не имеет смысла ссылаться на 1-й, 10-й или *n*-й элемент множества — вы можете только узнать, содержится ли элемент во множестве или нет. В со-

ответствии с этой функцией математических множеств множества .NET не поддерживают индексированный доступ, поэтому `ISet<T>` не требует поддержки `IList<T>`. В своей реализации `IEnumerable<T>` множества вольны выдавать свои члены в любом порядке, в котором пожелают.

Библиотека классов .NET предлагает два класса, которые используют этот интерфейс, с разными стратегиями реализации: `HashSet` и `SortedSet`. Как вы уже могли догадаться по именам, одна из двух встроенных реализаций множества на самом деле предпочитает держать свои элементы в порядке. `SortedSet` постоянно хранит свое содержимое отсортированным и представляет элементы в этом порядке через свою реализацию `IEnumerable<T>`. Документация не описывает точную стратегию, используемую для поддержания сортировки, но, по-видимому, использует сбалансированное двоичное дерево для поддержки эффективной вставки, удаления и обеспечения быстрого поиска при попытке определить, содержится ли в списке конкретное значение.

Другая реализация, `HashSet`, работает больше как `Dictionary< TKey, TValue >`. Он использует поиск на основе хеша, который часто бывает быстрее, чем упорядоченный подход, но, если вы захотите перебрать коллекцию с помощью `foreach`, результаты будут выдаваться беспорядочно. (Таким образом, отношения между `HashSet` и `SortedSet` очень похожи на отношения между словарем на основе хеша и упорядоченными словарями.)

Очереди и стеки

Очередь – это список, в который добавлять элементы можно только в конец, а удалять только первый элемент (после чего второй элемент, если он был, становится новым первым). Этот стиль списка часто называют списком по принципу «первым пришел, первым ушел» (FIFO). Это делает его менее полезным, чем `List<T>`, так как в случае `List<T>` вы можете читать, записывать, вставлять или удалять элементы в любой точке. Однако ограничения позволяют реализовать очередь со значительно лучшими показателями производительности для вставки и удаления. Когда вы удаляете элемент из `List<T>`, он должен сдвинуть все элементы после удаления, чтобы закрыть промежуток, и подобного же сдвига требует вставка. Вставка и удаление в конце `List<T>` эффективнее, но, если вам нужна семантика FIFO, вы не сможете всегда работать в конце — вам придется делать вставки или удаления в начале, что делает `List<T>` плохим выбором. `Queue<T>` использует гораздо более эффективную стратегию, поскольку он должен поддерживать только

семантику очереди. (Внутри используется кольцевой буфер, хотя это и не-документированная деталь реализации.)

Чтобы добавить новый элемент в конец очереди, нужно вызвать метод `Enqueue`. Чтобы удалить элемент в начале очереди, вызовите `Dequeue` или используйте `Peek`, если вы хотите посмотреть на элемент, не удаляя его. Обе операции приведут к исключению `InvalidOperationException`, если очередь пуста. Количество элементов в очереди можно узнать, используя свойство `Count`.

Хотя вы не можете вставлять, удалять или изменять элементы в середине списка, вы можете просмотреть всю очередь, потому что `Queue<T>` реализует `IEnumerable<T>`, а также предоставляет метод `ToArrayList`, который возвращает массив, содержащий копию текущего содержимого очереди.

Стек похож на очередь, за исключением того, что вы извлекаете элементы с того же конца, в который добавляете, так что этот список работает по принципу «последний пришел, первый вышел» (`LIFO`). `Stack<T>` выглядит очень похоже на `Queue<T>`, за исключением того, что вместо `Enqueue` и `Dequeue` методы добавления и удаления элементов используют традиционные имена для операций со стеком: `Push` и `Pop`. (Другие методы, такие как `Peek`, `ToArrayList` и т. д., остаются теми же.)

Библиотека классов не предлагает двустороннюю очередь (поэтому не существует эквивалента классу `deque` в C++). Тем не менее расширенный набор этих функций предлагают связанные списки.

Связные списки

Класс `LinkedList<T>` обеспечивает реализацию классической структуры данных двусвязного списка, в котором каждый элемент в последовательности заключен в объект (типа `LinkedListNode<T>`), который предоставляет ссылки на предшествующий и следующий элементы.

Преимущество связного списка состоит в том, что вставка и удаление являются экономичными — для них не требуется перемещать элементы в массивах, как не требуется и балансировка двоичных деревьев. Ему просто нужно поменять местами пару ссылок. Недостатком является то, что связанные списки имеют довольно большие накладные расходы в плане памяти, что требует дополнительного объекта в куче для каждого элемента

в коллекции. Кроме того, для ЦП относительно накладно добираться до n -го элемента, потому что вы должны перейти к началу, а затем пройти n узлов.

Первый и последний узлы в `LinkedList<T>` доступны через свойства с предсказуемым именем `First` и `Last`. Вы можете вставить элементы в начало или конец списка с помощью `AddFirst` и `AddLast` соответственно. Чтобы добавить элементы в середину списка, вызовите `AddBefore` или `AddAfter`, передав `LinkedListNode<T>`, до или после которого вы хотите добавить новый элемент.

Список также содержит методы `RemoveFirst`, `RemoveLast` и две перегрузки метода `Remove`, которые позволяют удалить либо первый узел с указанным значением, либо определенный `LinkedListNode<T>`.

Сам `LinkedListNode<T>` предоставляет свойство `Value` типа `T`, содержащее фактический элемент для данной позиции узла в последовательности. Его свойство `List` ссылается на содержащий `LinkedList<T>`, а свойства `Previous` и `Next` позволяют найти предыдущий или следующий узел.

Чтобы перебрать содержимое связного списка, вы, конечно, можете извлечь первый узел из свойства `First`, а затем вызывать свойство `Next` каждого узла, пока не получите `null`. Однако `LinkedList<T>` реализует `IEnumerable<T>` так, что проще будет использовать цикл `foreach`. Если вы хотите получить элементы в обратном порядке, начните с `Last` и вызывайте `Previous` для каждого следующего узла. Если список пуст, `First` и `Last` вернут `null`.

Параллельные коллекции

Описываемые до сих пор классы коллекций предназначены для однопоточного использования. Вы можете использовать разные экземпляры в разных потоках одновременно, но конкретный экземпляр любого из этих типов должен использоваться только из одного потока одновременно. Но некоторые типы предназначены для одновременного использования несколькими потоками, без необходимости использовать механизмы синхронизации, описанные в главе 16⁸. Они находятся в пространстве имен `System.Collections.Concurrent`.

Параллельные коллекции не предлагают эквивалентов для каждого непараллельного типа коллекций. Некоторые классы предназначены для решения

⁸ Из этого правила есть исключение: вы можете использовать коллекцию из нескольких потоков, если ни один из потоков не пытается изменить ее.

конкретных проблем параллельного программирования. Но даже в случае тех, которые имеют непараллельные аналоги, необходимость одновременного использования без блокировки может означать, что они представляют несколько иной API, чем любой из обычных классов коллекций.

Классы `ConcurrentQueue<T>` и `ConcurrentStack<T>` больше всего похожи на непараллельные коллекции, которые мы уже видели, хотя они им не идентичны. `Dequeue` и `Peek` очереди были заменены `TryDequeue` и `TryPeek`, потому что в параллельном окружении нет надежного способа заранее узнать, будет ли попытка получить элемент из очереди успешной. (Вы можете проверить счетчик очереди, но даже если он не равен нулю, другой поток может зайти туда и очистить очередь между проверкой счетчика и попыткой получить элемент.) Таким образом, операция получения элемента должна быть атомарной и с проверкой доступности элемента, отсюда и формы `Try`, которые могут быть неудачными без вызова исключения. Аналогичным образом параллельный стек предоставляет `TryPop` и `TryPeek`.

`ConcurrentDictionary< TKey, TValue >` выглядит довольно похожим на своего двоюродного брата, но добавляет ряд дополнительных методов для обеспечения атомарности, требуемой в параллельном окружении: метод `TryAdd` объединяет проверку на наличие ключа с добавлением новой записи; `GetOrAdd` делает то же самое, но дополнительно возвращает существующее значение, если оно есть, как часть одной и той же атомарной операции.

Параллельного списка не существует, поскольку для успешного использования упорядоченных, проиндексированных списков в параллельном окружении вам, как правило, требуется более грубая синхронизация. Но если вам просто требуется куча объектов, можно воспользоваться `ConcurrentBag<T>`, который не поддерживает какую-либо конкретную упорядоченность.

Также имеется `BlockingCollection<T>`, который работает как очередь, но позволяет потокам, которые хотят удалить элементы из очереди, выбрать блокировку, пока элемент не станет доступным. Вы также можете установить ограниченную емкость и создать потоки, которые, если очередь заполнена, помещают элементы в блок, ожидая, пока освободится место.

Неизменяемые коллекции

Microsoft предоставляет набор классов коллекций, которые гарантируют неизменность и в то же время предоставляют легкий способ создать мани-

фицированную версию коллекции без необходимости делать полную копию. В отличие от типов, обсуждаемых в этой главе, они не встроены в ту часть библиотеки классов, которая поставляется с .NET, поэтому им необходимо ссылаться на пакет NuGet `System.Collections.Immutable`.

Неизменность может быть очень полезной характеристикой в многопоточных средах, потому что если вы знаете, что данные, с которыми вы работаете, не могут быть изменены, вам не нужно предпринимать особых мер предосторожности для синхронизации доступа к ним. (Это более надежная гарантия, чем та, что вы получаете с `IReadOnlyList<T>`, который просто мешает вам изменить коллекцию; он может быть просто фасадом коллекции, которую может изменить какой-то другой поток.) Но что вам делать, если ваши данные нужно периодически обновлять? Будет жалко отказываться от неизменности и терпеть издержки традиционной многопоточной синхронизации в тех случаях, когда вы ожидаете, что конфликты будут редкими.

Низкотехнологичный подход заключается в создании новой копии всех ваших данных каждый раз, когда что-то меняется (например, когда вы хотите добавить элемент в коллекцию, создайте целую новую коллекцию с копией всех старых элементов и новым и дальше используйте эту новую коллекцию). Это работает, но может оказаться крайне неэффективным решением. Однако существуют технологии, которые по сути могут повторно использовать части существующих коллекций. Основной принцип заключается в том, что если вы хотите добавить элемент в коллекцию, то создаете новую коллекцию, которая просто указывает на уже существующие данные вместе с некоторой дополнительной информацией, указывающей на изменения. На практике это несколько сложнее, но суть заключается в том, что существуют хорошо проработанные способы реализации различных видов коллекций, позволяющие успешно создавать то, что выглядит как полные автономные копии исходных данных с небольшими изменениями. При этом нет необходимости изменения исходных данных или создания полной копии коллекции. Неизменяемые коллекции делают все это за вас, инкапсулируя работу в понятные интерфейсы.

Это допускает схему, при которой вы можете обновлять модель вашего приложения, не затрагивая код, который был в процессе использования текущей версии данных. Следовательно, вам не нужно держать блокировки во время чтения данных — вам может потребоваться некоторая синхронизация при получении последней версии данных, но после этого вы можете обрабатывать данные без каких-либо проблем с параллелизмом. Это может оказаться

особенно полезно при написании многопоточного кода. Платформа компилятора .NET (часто известная под кодовым названием Roslyn), которая является основой компилятора Microsoft C#, использует эту технику, чтобы компиляция могла эффективно использовать несколько процессорных ядер.

Пространство имен `System.Collections.Immutable` определяет свои собственные интерфейсы — `IImmutableList<T>`, `IImmutableDictionary< TKey, TValue >`, `IImmutableQueue<T>`, `IImmutableStack<T>` и `IImmutableSet<T>`. Это необходимо, потому что все операции, которые каким-либо образом изменяют коллекцию, должны возвращать новую коллекцию. Листинг 5.49 показывает, что это означает для операции добавления записей в словарь.

Листинг 5.49. Создание неизменяемых словарей

```
IImmutableDictionary <int, string> d =
    ImmutableDictionary.Create <int, string> ();
d = d.Add(1, "One");
d = d.Add(2, "Two");
d = d.Add(3, "Three");
```

Весь смысл неизменяемых типов заключается в том, что код, использующий существующий объект, может быть уверен, что ничего не изменится. Именно поэтому добавления, удаления или модификации обязательно означают создание нового объекта, который будет выглядеть точно так же, как старый, но с примененным изменением. (Встроенный тип `string` работает точно так же, потому что он также неизменен — методы, которые выглядят так, как будто они изменят значение, например `Trim`, фактически возвращают новую строку.) Так что в листинге 5.49 переменная `d` последовательно ссылается на четыре разных неизменяемых словаря: пустой, с одним значением, с двумя значениями и, наконец, со всеми тремя значениями.

Если вы добавляете подобный диапазон значений и не планируете делать промежуточные результаты доступными для другого кода, более эффективно добавлять несколько значений в одной операции, поскольку для этого не нужно создавать отдельный объект `IImmutableDictionary< TKey, TValue >` для каждой добавляемой записи. (Вы можете сравнить работу неизменяемых коллекций с системой управления исходным кодом, причем каждое изменение работает как фиксация изменений — для каждой фиксации изменений, которую вы делаете, будет существовать версия коллекции, которая представляет свое содержимое сразу после этого изменения.) Более эффективно объединять несколько взаимосвязанных изменений в одну «версию», по-

этому все коллекции содержат методы `AddRange`, позволяющие добавлять несколько элементов за раз.

При создании новой коллекции с нуля применяется тот же принцип: эффективнее будет, если вы поместите все изначальное содержимое в первую версию коллекции вместо добавления элементов по одному. Для упрощения этой задачи каждый тип неизменяемой коллекции предлагает вложенный класс `Builder`, позволяющий добавлять элементы по одному, но откладывать фактическое создание коллекции до завершения. Листинг 5.50 показывает, как это делается.

Листинг 5.50. Создание неизменяемого словаря с использованием построителя

```
ImmutableDictionary<int, string>.Builder b =  
    ImmutableDictionary.CreateBuilder<int, string>();  
b.Add(1, "One");  
b.Add(2, "Two");  
b.Add(3, "Three");  
IImmutableDictionary<int, string> d = b.ToImmutable();
```

Объект построителя не является неизменяемым. Как и `StringBuilder`, это изменяемый объект, который предоставляет эффективный способ построения описания неизменяемого объекта.

ImmutableArray<T>

В дополнение к неизменяемым списку, словарю, очереди, стеку и типам множеств есть еще один неизменяемый класс коллекции, который немного отличается от остальных: `ImmutableArray<T>`. По сути, это обертка, строящая вокруг массива неизменяемый фасад. Он реализует `IImmutableList<T>`, и это означает, что он предоставляет те же возможности, что и неизменяемый список, но у него совсем другие характеристики производительности.

Когда вы вызываете `Add` в неизменяемом списке, он пытается повторно использовать большую часть существующих данных, поэтому если у вас в списке миллион элементов, «новый» список, возвращаемый `Add`, не будет содержать новую копию этих элементов — в основном он будет использовать данные, которые уже были там ранее. Однако чтобы достичь этого, `ImmutableList<T>` использует достаточно сложную внутреннюю древовидную структуру данных. В результате поиск значений по индексу в `ImmutableList<T>` далеко не так эффективен, как использование массива (или `List<T>`). Индексатор для `ImmutableList<T>` имеет сложность $O(\log n)$.

`ImmutableArray<T>` гораздо более эффективен в случае чтения, так как является оберткой вокруг массива и имеет сложность $O(1)$, т. е. время, необходимое для извлечения записи, является постоянным независимо от размера коллекции. Компромисс состоит в том, что все методы `IImmutableList<T>` для создания измененной версии списка (`Add`, `Remove`, `Insert`, `SetItem` и т. д.) создают полностью новый массив, включая новую копию любых данных, которые необходимо перенести. (Другими словами, в отличие от всех остальных неизменяемых типов коллекций, `ImmutableArray<T>` использует низкотехнологичный подход для обеспечения неизменяемости, который я описал ранее.) Это делает модификации намного более дорогостоящими, но если у вас есть данные, которые будут либо изменяться редко, либо вообще не изменяться после первоначального создания массива, то это отличный компромисс, потому что будет создаваться только одна копия массива.

Итог

В этой главе мы рассмотрели предоставляемую средой выполнения внутреннюю поддержку массивов, а также различные классы коллекций, которые предлагает .NET, если вам нужно нечто большее, чем список элементов фиксированного размера. Далее мы обратимся к более сложной теме — наследованию.

ГЛАВА 6

Наследование

Классы C# поддерживают наследование — популярный объектно ориентированный механизм повторного использования кода. При написании класса вы можете указать базовый класс. Ваш класс будет происходить от него, и это означает, что все, что присутствует в базовом классе, будет присутствовать и в вашем, вместе со всеми новыми членами, которые вы добавляете.

Классы поддерживают только одиночное наследование (поэтому можно указать только один базовый класс). Чего-то похожего на множественное наследование можно добиться с помощью интерфейсов. Значимые типы вообще не поддерживают наследование. Одна из причин этого заключается в том, что значимые типы обычно не используются по ссылке, что устраняет одно из основных преимуществ наследования: полиморфизм во время выполнения. Наследование не обязательно несовместимо с поведением значимых типов — некоторые языки с этим справляются, но это часто приводит к проблемам. Например, задание значения некоторого производного типа в переменную его базового типа приводит к потере всех полей, добавленных производным типом, и эта проблема называется *нарезкой* (*slicing*). С# обходит ее, ограничивая наследование ссылочными типами. Когда вы присваиваете переменной базового типа переменную производного, то вы копируете ссылку, а не сам объект, поэтому объект остается неизменным. Нарезка является проблемой, только когда базовый класс содержит метод, который клонирует объект, и не предоставляет возможность для его расширения в производных классах (или такой способ есть, но производный класс не делает этого).

Классы указывают базовый класс с помощью синтаксиса, показанного в листинге 6.1, — базовый тип записывается после двоеточия, следующего за именем класса. В этом примере предполагается, что класс `SomeClass` был определен где-то в другом месте проекта или в одной из используемых библиотек.

Как вы уже видели в главе 3, если класс реализует какие-либо интерфейсы, они также перечисляются после двоеточия. Если вам одновременно нужно и наследовать, и реализовать интерфейсы, то базовый класс должен указываться первым, как это сделано во втором классе в листинге 6.1.

Листинг 6.1. Указание базового класса

```
public class Derived : SomeClass
{
}

public class AlsoDerived : SomeClass, IDisposable
{
    public void Dispose() { }
}
```

Вы можете наследовать от класса, который, в свою очередь, наследуется от другого класса. Класс `MoreDerived` в листинге 6.2 является производным от `Derived`, который, в свою очередь, является производным от `Base`.

Листинг 6.2. Цепочка наследования

```
public class Base
{
}

public class Derived : Base
{
}

public class MoreDerived : Derived
{
}
```

Это означает, что технически `MoreDerived` имеет несколько базовых классов: он является производным от `Derived` (напрямую) и `Base` (косвенно через `Derived`). Это не множественное наследование, поскольку есть только одна цепочка наследования — любой отдельный класс наследуется не более чем от одного базового класса. (Все классы прямо или косвенно происходят от `object`, который становится базовым классом по умолчанию, если вы его не указали.)

Поскольку производный класс наследует все, что имеет базовый класс, — все его поля, методы и другие члены, как открытые, так и закрытые, — экземпляр производного класса может делать все, что делает экземпляр базового

класса. Это классические отношения «is-a»¹, которые во многих языках подразумеваются под наследованием. Любой экземпляр `MoreDerived` — это `Derived`, а также `Base`. Система типов C# распознает именно эту связь.

Наследование и преобразования

C# обеспечивает различные встроенные неявные преобразования. В главе 2 мы видели преобразования числовых типов, но возможны преобразования и ссылочных типов. Если некоторый тип `D` происходит от `B` (прямо или косвенно), то ссылка типа `D` может быть неявно преобразована в ссылку типа `B`. Это следует из отношения «is-a», которые я описал в предыдущем разделе, а именно любой экземпляр `D` — это `B`. Это неявное преобразование включает полиморфизм: код, написанный для работы с `B`, сможет работать и с производным от `B`.



Преобразования ссылок работают особым образом. В отличие от других преобразований, они никак не меняют значение. (Все встроенные неявные числовые преобразования создают новое значение из своих входных данных, часто изменяя само представление. Например, двоичное представление целого числа 1 выглядит по-разному для типов `float` и `int`.) По сути, они преобразуют интерпретацию ссылки, а не саму ссылку или объект, на который она указывает. Как вы увидите позже в этой главе, в различных местах CLR учитывает наличие неявного преобразования ссылок, но не других форм преобразования.

Очевидно, что неявного преобразования в обратном направлении не существует — хотя переменная типа `B` могла бы ссылаться на объект типа `D`, но нет гарантии, что она будет это делать. Из типа `B` можно получить любое количество типов, и переменная `B` может ссылаться на экземпляр любого из них. Тем не менее иногда вам может понадобиться преобразовать ссылку из базового типа в производный, и эта операция иногда называется нисходящим преобразованием. Например, вы точно знаете, что конкретная переменная содержит ссылку определенного типа. Или, возможно, вы не уверены и хотели бы, чтобы ваш код предоставлял дополнительные возможности для определенных типов. C# предлагает четыре способа сделать это.

¹ Отношение подчинения между абстракциями, т. е. класс является подклассом другого класса. — Примеч. ред.

Мы можем попытаться выполнить нисходящее приведение типа с использованием синтаксиса приведения. Это тот же синтаксис, который используется для выполнения неявных числовых преобразований, как показано в листинге 6.3.

Листинг 6.3. Нисходящее приведение типа

```
public static void UseAsDerived(Base baseArg)
{
    var d = (Derived) baseArg;
    // ... делаем что-то с d
}
```

Успех такого преобразования не гарантирован, и именно поэтому мы не можем использовать неявное преобразование. Если вы попытаетесь это сделать, когда аргумент `baseArg` ссылается на что-то, что не является ни экземпляром `Derived`, ни чем-то производным от `Derived`, преобразование не сработает и возникнет исключение `InvalidCastException`. (Исключения описаны в главе 8.)

Таким образом, приведение целесообразно только в том случае, если вы уверены, что объект действительно соответствует ожидаемому вами типу, и вы сочли бы ошибкой, если бы оказалось, что это не так. Это полезно, когда API принимает объект, который он позже вам вернет. Так поступают многие асинхронные API, потому что в случаях, когда вы запускаете несколько операций одновременно, вам нужен какой-то способ выяснить, какая именно операция завершилась при получении уведомления о завершении (хотя, как мы увидим в следующих главах, существуют различные решения этой проблемы). Поскольку эти API не знают, какой тип данных вы хотите связать с операцией, они обычно просто принимают ссылку на тип `object` и вы обычно используете приведение, чтобы превратить его в ссылку на требуемый тип, когда ссылка в конечном итоге возвращается к вам.

Не всегда можно знать наверняка, имеет ли объект определенный тип. В этом случае используйте оператор `as`, как показано в листинге 6.4. Он позволит выполнить преобразование, не рискуя получить исключение. Если преобразование не удастся, этот оператор просто возвращает `null`.

Хотя этот метод довольно часто встречается в уже существующем коде, введение шаблонов в C# 7.0 дало более лаконичную альтернативу. Лис-

тинг 6.5 работает так же, как и листинг 6.4: тело `if` запускается, только если `b` ссылается на экземпляр `Derived`, и в этом случае к нему можно получить доступ через переменную `d`. Ключевое слово `is` здесь указывает на то, что мы хотим сравнить `b` с шаблоном. В этом случае мы используем шаблон типа, который выполняет тот же тип проверки во время выполнения, что и оператор `as`. Выражение, которое применяет шаблон с помощью `is`, возвращает `bool`, указывающий, было ли совпадение с шаблоном. Мы можем использовать этот результат как выражение условия оператора `if`, что избавляет от необходимости сравнения с `null`. А поскольку шаблоны типов включают в себя объявление и инициализацию переменных, работа, для которой потребовались два оператора в листинге 6.4, может быть помещена в оператор `if` в листинге 6.5.

Листинг 6.4. Оператор `as`

```
public static void MightUseAsDerived(Base b)
{
    var d = b as Derived;

    if (d != null)
    {
        // ... делаем что-то с d
    }
}
```

Листинг 6.5. Шаблон типа

```
public static void MightUseAsDerived(Base b)
{
    if (b is Derived d)
    {
        // ... делаем что-то с d
    }
}
```

Помимо того что оператор `is` является более компактным, он также имеет преимущество работы в одном сценарии, недоступном для `as`: вы можете проверить, указывает ли ссылка на тип `object` на экземпляр значимого типа, такой как `int`. (На первый взгляд это выглядит противоречием — как можно иметь ссылку на то, что не является ссылочным типом? Глава 7 покажет, как это возможно.) Оператор `as` не будет работать, потому что он возвращает `null`, когда экземпляр не принадлежит указанному типу, и, конечно, он не может проделать это со значимым типом — такого понятия, как `null` типа `int`,

не существует. Поскольку шаблон типа устраниет необходимость проверки на `null`, мы просто используем результат `bool`, который выдает оператор `is`, так как можем использовать значимые типы.

Наконец, иногда может быть полезно узнать, указывает ли ссылка на объект определенного типа, при этом нам не нужно использовать члены, специфичные для этого типа. Например, для определенного производного класса вы можете пропустить какой-то конкретный фрагмент обработки. Для этого мы также можем использовать оператор `is`. Если вы просто поместите имя типа вместо полного шаблона, как в листинге 6.6, он проверит, принадлежит ли объект определенному типу, и вернет `true`, если это так, и `false` в противном случае.

Листинг 6.6. Оператор `is`

```
if (!(b is WeirdType))
{
    // ... выполнить обработку, кроме той, которую требует WeirdType
}
```

Форма оператора `is` представляет собой исторический курьез. Его использование основано на шаблонах, но это не шаблон. (Вы не можете написать просто имя типа в любом другом месте, где нужен шаблон.) И он избыточен: такого же эффекта можно достичь с шаблоном типа, который сбрасывает свой вывод. (Версия, основанная на шаблонах, будет выглядеть как `!(b is WeirdType _)`.) Причина, по которой существует эта не основанная на шаблонах версия, — до этого она долго была единственным способом. Шаблоны были добавлены в C# только в версии 7.0, тогда как `is` был в языке с самого начала.

При конвертации с использованием только что описанных методов вам не обязательно указывать точный тип. Эти операции будут успешными, пока возможно неявное преобразование ссылки из реального типа объекта в тип, который вы ищете. Например, имея типы `Base`, `Derived` и² `MoreDerived`, которые определены в листинге 6.2, предположим, что у вас имеется переменная типа `Base`, которая в настоящее время содержит ссылку на экземпляр `MoreDerived`. Очевидно, что вы могли бы привести ссылку на `MoreDerived` (для этого типа сработает и `as`, и `is`), но, как вы, вероятно, понимаете, преобразование в `Derived` тоже будет работать.

² Это исключает пользовательские неявные преобразования.

Эти четыре механизма работают и в случае интерфейсов. Когда вы попытаетесь преобразовать ссылку в ссылку на тип интерфейса (или проверите на тип интерфейса с помощью `is`), у вас это получится, если указанный объект реализует соответствующий интерфейс.

Наследование интерфейса

Интерфейсы поддерживают наследование, но это не совсем то же самое, что наследование классов. Синтаксис похож, но, как показано в листинге 6.7, интерфейс может иметь несколько базовых интерфейсов. В то время как .NET предлагает только одно наследование реализации, это ограничение не применяется к интерфейсам, поскольку большинство сложностей и потенциальных двусмысленностей, которые могут возникнуть при множественном наследовании, не касаются чисто абстрактных типов. Даже добавление реализаций интерфейса по умолчанию в C# 8.0 этого не изменило, потому что они по-прежнему не могут добавлять поля или открытые члены к реализуемому типу. (Когда класс использует для члена реализацию по умолчанию, этот член доступен только через ссылки типа интерфейса.)

Листинг 6.7. Наследование интерфейса

```
interface IBase1
{
    void Base1Method();
}

interface IBase2
{
    void Base2Method();
}

interface IBoth : IBase1, IBase2
{
    void Method3();
}
```

Хотя *наследование интерфейса* является официальным названием этой функции, оно не является правильным. Тогда как производные классы наследуют все члены базовых, производные интерфейсы — нет. Может показаться, что они это делают, — имея переменную типа `IBoth`, вы можете вызывать методы `Base1Method` и `Base2Method`, определенные ее базовыми

интерфейсами. Однако истинное значение наследования интерфейса заключается в том, что тип, реализующий интерфейс, обязан реализовывать все унаследованные интерфейсы. Таким образом, класс, который реализует `IBoth`, должен также реализовать `IBase1` и `IBase2`. Это тонкое различие, тем более что C# не требует явного перечисления базовых интерфейсов. Класс в листинге 6.8 только объявляет, что он реализует `IBoth`. Но если вы используете API отражения .NET для проверки определения типа, вы обнаружите, что компилятор добавил `IBase1` и `IBase2` в список интерфейсов, которые реализует класс, как и явно объявленный `IBoth`.

Листинг 6.8. Реализация производного интерфейса

```
public class Impl : IBoth
{
    public void Base1Method()
    {
    }

    public void Base2Method()
    {
    }

    public void Method3()
    {
    }
}
```

Поскольку реализации производного интерфейса должны реализовывать все базовые интерфейсы, C# позволяет получать доступ к членам базовых интерфейсов напрямую через ссылку на производный тип, поэтому переменная типа `IBoth` обеспечивает доступ к `Base1Method` и `Base2Method`, а также к собственному методу `Method3`. Есть неявные преобразования из производных типов интерфейса в их базовые типы. Например, ссылка типа `IBoth` может быть присвоена переменным типа `IBase1` и `IBase2`.

Обобщения

Если вы наследуете от обобщенного класса, вы должны указать требуемые аргументы типа. Необходимо предоставить конкретные типы, если ваш производный тип не является обобщенным, т. е. может использовать свои собственные параметры типа в качестве аргументов. В листинге 6.9 пока-

заны оба метода, как и то, что при наследовании от класса с несколькими параметрами типа можно использовать смешанный подход, указав один аргумент типа напрямую, а другой транслировав.

Листинг 6.9. Наследование от обобщенного базового класса

```
public class GenericBase1<T>
{
    public T Item { get; set; }
}

public class GenericBase2< TKey, TValue >
{
    public TKey Key { get; set; }
    public TValue Value { get; set; }
}

public class NonGenericDerived : GenericBase1<string>
{

}

public class GenericDerived<T> : GenericBase1<T>
{

}

public class MixedDerived<T> : GenericBase2<string, T>
{}
```

Хотя вы можете использовать любой из ваших параметров типа в качестве аргументов типа для базового класса, вы не можете наследовать от параметра типа. Это немного разочаровывает, если вы привыкли к языкам, которые подобное допускают, но спецификация языка C# это просто запрещает. Однако вам разрешено использовать ваш собственный тип в качестве аргумента типа для вашего базового класса. И вы также можете задать ограничение для аргумента типа, требующее его наследования от вашего собственного типа. Листинг 6.10 все это показывает.

Листинг 6.10. Требование наследования аргумента типа от типа, к которому он применяется

```
public class SelfAsTypeArgument : IComparable<SelfAsTypeArgument>
{
    // ... реализация удалена для ясности
}
```

```
public class Curious<T>
    where T : Curious<T>
{
}
```

Ковариантность и контравариантность

В главе 4 я упомянул, что обобщенные типы имеют особые правила совместимости типов, называемые ковариантностью и контравариантностью. Эти правила определяют, являются ли ссылки на определенные обобщенные типы неявно конвертируемыми друг в друга, когда имеются неявные преобразования между их аргументами типов.



Ковариантность и контравариантность применимы только к обобщенным аргументам типа интерфейсов и делегатов. (Делегаты описаны в главе 9.) Вы не можете определить ковариантный или контравариантный класс или структуру.

Рассмотрим простые классы `Base` и `Derived`, ранее показанные в листинге 6.2, и метод в листинге 6.11, который принимает любой `Base`. (Он ничего с ним не делает, но это несущественно. Важно то, что его сигнатура говорит, что он может его использовать.)

Листинг 6.11. Метод, принимающий любой Base

```
public static void UseBase(Base b)
{
}
```

Мы уже знаем, что помимо принятия ссылки на любой `Base` он также может принимать ссылку на экземпляр любого типа, производного от `Base`, например `Derived`. Учитывая это, рассмотрим метод в листинге 6.12.

Листинг 6.12. Метод, принимающий любой `IEnumerable<Base>`

```
public static void AllYourBase(IEnumerable<Base> bases)
{
}
```

Для него требуется объект, который реализует `IEnumerable<T>`, обобщенный интерфейс из главы 5, где `T` — это `Base`. Что бы вы ожидали увидеть, попытайся мы передать объект, который не реализовал `IEnumerable<Base>`,

но реализовал `IEnumerable<Derived>`? Листинг 6.13 делает это и отлично компилируется.

Листинг 6.13. Передача `IEnumerable<T>` производного типа

```
IEnumerable<Derived> derivedItems =
    new Derived[] { new Derived(), new Derived() };
AllYourBase(derivedItems);
```

Интуитивно мы понимаем, что тут есть смысл. Метод `AllYourBase` ожидает объект, который может предоставить последовательность объектов, имеющих тип `Base`. `IEnumerable<Derived>` подходит, потому что предоставляет последовательность объектов `Derived`, а любой объект `Derived` также является `Base`. А что насчет кода из листинга 6.14?

Листинг 6.14. Метод, принимающий любой `ICollection<Base>`

```
public static void AddBase(ICollection<Base> bases)
{
    bases.Add(new Base());
}
```

Из главы 5 мы помним, что `ICollection<T>` происходит от `IEnumerable<T>` и добавляет определенные способы изменения коллекции. Этот конкретный метод использует это, добавляя в коллекцию новый объект `Base`. Это привело бы к проблемам для кода в листинге 6.15.

Листинг 6.15. Ошибка: попытка передать `ICollection<T>` с производным типом

```
ICollection<Derived> derivedList = new List<Derived>();
AddBase(derivedList); // Не скомпилируется
```

Код, использующий переменную `derivedList`, будет ожидать, что каждый объект в этом списке будет иметь тип `Derived` (или производный от него, например класс `MoreDerived` из листинга 6.2). Но метод `AddBase` в листинге 6.14 пытается добавить простой экземпляр `Base`. Это не может быть правильно, и компилятор не позволяет этого сделать. Вызов `Add Base` вызовет ошибку компилятора с жалобой на то, что ссылки типа `ICollection<Derived>` нельзя неявно преобразовать в ссылки типа `ICollection<Base>`.

Откуда компилятор знает, что этого делать нельзя, в то время как очень похожее преобразование из `IEnumerable<Derived>` в `IEnumerable<Base>` разрешено? Кстати, это происходит не потому, что листинг 6.14 содержит проблемный код. Вы получите точно такую же ошибку компилятора, если метод

`AddBase` будет полностью пустым. Причина, по которой мы не получаем ошибку в листинге 6.13, заключается в том, что интерфейс `IEnumerable<T>` объявляет свой аргумент типа `T` как ковариантный. Синтаксис для этого вы уже видели в главе 5, но тогда я не обратил на него вашего внимания, поэтому в листинге 6.16 еще раз показана соответствующая часть из определения этого интерфейса.

Листинг 6.16. Ковариантный параметр типа

```
public interface IEnumerable<out T> : IEnumerable
```

Всю работу берет на себя ключевое слово `out`. (Опять же, C# соблюдает традицию семейства C, согласно которой каждому ключевому слову присваивается несколько задач. Впервые мы увидели это ключевое слово в контексте параметров метода, которые могут возвращать информацию вызывающей стороне.) Интуиция подсказывает, что описание аргумента типа `T` как `out` оправдано в том смысле, что интерфейс `IEnumerable<T>` лишь предоставляет `T`, но не определяет каких-либо членов, которые его принимают. (Интерфейс использует этот параметр типа только в одном месте: в своем свойстве только для чтения `Current`.)

Сравните это с `ICollection<T>`, который происходит от `IEnumerable<T>`. Вполне очевидно, что из него можно получить `T`, но также можно и передать `T` в его метод `Add`. Следовательно, `ICollection<T>` не может снабдить свой аргумент типа ключевым словом `out`. (Если вы попытаетесь написать свой собственный подобный интерфейс, компилятор выдаст ошибку, если вы объявили аргумент типа ковариантным. Он не будет верить вам на слово, а проверит, что вы действительно никуда не можете передать `T`.)

Компилятор отклонит код в листинге 6.15, потому что `T` не является ковариантным в `ICollection<T>`. Понятия *ковариантности* и *контравариантности* происходят из раздела математики под названием *теория категорий*³. Параметры, которые ведут себя как `T` из `IEnumerable<T>`, называются ковариантными, потому что неявные ссылочные преобразования для обобщенного типа работают в том же направлении, что и преобразования для аргумента типа: *Derived* неявно конвертируется в *Base*, а поскольку `T` является ковариантным в `IEnumerable<T>`, то `IEnumerable<Derived>` неявно конвертируется в `IEnumerable<Base>`.

³ Теория категорий — раздел математики, изучающий свойства отношений между математическими объектами, не зависящие от внутренней структуры объектов. — Примеч. ред.

Контравариантность вполне ожидаемо работает наоборот, и, как вы, наверное, догадались, будет обозначена ключевым словом `in`. Проще всего увидеть ее в действии в коде, который использует члены типов, поэтому в листинге 6.17 показана несколько более интересная пара классов, чем в предыдущих примерах.

Листинг 6.17. Иерархия классов с фактическими членами

```
public class Shape
{
    public Rect BoundingBox { get; set; }

public class RoundedRectangle : Shape
{
    public double CornerRadius { get; set; }
}
```

Листинг 6.18 определяет два класса, которые используют эти типы фигур. Оба реализуют `IComparer<T>`, который я показал в главе 4. `BoxAreaComparer` сравнивает две фигуры на основе площадей их ограничивающих прямоугольников — фигура, чей ограничивающий прямоугольник покрывает большую площадь, будет в данном сравнении считаться большей. С другой стороны, `CornerSharpnessComparer` сравнивает скругленные прямоугольники, оценивая то, насколько заострены их углы.

Ссылки типа `RoundedRectangle` неявно преобразуются в `Shape`. А что насчет `IComparer<T>`? Наш `BoxAreaComparer` может сравнивать любые фигуры, о чем и заявляет, реализуя `IComparer<Shape>`. Аргумент типа `T` функции сравнения всегда используется только в методе `Compare`, и ему подходит любой `Shape`. Его не смутит, если мы передадим ему пару ссылок `RoundedRectangle`, так что наш класс вполне отвечает требованиям `IComparer<RoundedRectangle>`. Поэтому неявное преобразование из `IComparer<Shape>` в `IComparer<RoundedRectangle>` имеет смысл.

Листинг 6.18. Сравнение фигур

```
public class BoxAreaComparer : IComparer<Shape>
{
    public int Compare(Shape x, Shape y)
    {
        double xArea = x.BoundingBox.Width * x.BoundingBox.Height;
        double yArea = y.BoundingBox.Width * y.BoundingBox.Height;
```

```
        return Math.Sign(xArea - yArea);
    }
}

public class CornerSharpnessComparer : IComparer<RoundedRectangle>
{
    public int Compare(RoundedRectangle x, RoundedRectangle y)
    {
        // Меньшие углы острее, поэтому меньший радиус «больше»
        // в терминах этого сравнения, отсюда обратное вычитание.
        return Math.Sign(y.CornerRadius - x.CornerRadius);
    }
}
```

Однако класс `CornerSharpnessComparer` более изощренный. Он использует свойство `CornerRadius`, которое доступно только для скругленных прямоугольников, но не для старой доброй `Shape`. Поэтому из `IComparer<RoundedRectangle>` нет неявного преобразования в `IComparer<Shape>`.

Это противоположно тому, что мы видели в `IEnumerable<T>`. Неявное преобразование между `IEnumerable<T1>` и `IEnumerable<T2>` возможно, когда есть неявное преобразование ссылок из `T1` в `T2`. Но неявное преобразование между `IComparer<T1>` и `IComparer<T2>` доступно, когда неявное преобразование ссылок возможно в другом направлении: из `T2` в `T1`. Это обратное отношение называется контравариантностью. Листинг 6.19 является выдержкой из определения `IComparer<T>` и показывает этот контравариантный параметр типа.

Листинг 6.19. Контравариантный параметр типа

```
public interface IComparer<in T>
```

Большинство обобщенных параметров типа не являются ни ковариантными, ни контравариантными. (Они инвариантны.) `ICollection<T>` не может быть вариантым, потому что содержит некоторые члены, которые принимают `T`, и некоторые, которые его возвращают. `ICollection<Shape>` может содержать фигуры, которые не являются `RoundedRectangles`, поэтому вы не можете передать его методу, ожидающему `ICollection<RoundedRectangle>`. Такой метод будет ожидать, что каждый объект, который он извлекает из коллекции, будет закругленным прямоугольником. И наоборот, нельзя ожидать от `ICollection<RoundedRectangle>`, что в нее можно будет добавлять фигуры, отличные от скругленных прямоугольников. Поэтому вы не можете передать `ICollection<RoundedRec tangle>` в метод, который ожидает

`ICollection<Shape>`, так как этот метод может попытаться добавить другие виды фигур.

Массивы ковариантны, как и `IEnumerable<T>`. Это довольно странно, потому что мы можем писать такие методы, как показаны в листинге 6.20.

Листинг 6.20. Изменение элемента в массиве

```
public static void UseBaseArray(Base[] bases)
{
    bases[0] = new Base();
}
```

Если бы я вызвал это из кода в листинге 6.21, то совершил бы ту же ошибку, что и в листинге 6.15, где попытался передать `ICollection<Derived>` в метод, который хотел поместить в коллекцию что-то отличное от `Derived`. Хотя листинг 6.15 и не компилируется, листинг 6.21 это делает из-за неожиданной ковариации массивов.

Листинг 6.21. Передача массива с производным типом элемента

```
Derived[] derivedBases = { new Derived(), new Derived() };
UseBaseArray(derivedBases);
```

Это выглядит так, как будто мы можем схитрить и заставить массив принимать ссылку на объект, который не является экземпляром типа элемента массива, в данном случае помещая ссылку на объект `Base` (который не является `Derived`) в `Derived[]`. Но это было бы нарушением системы типов. Значит ли это, что небеса обрушатся?

C#, как и положено, запрещает подобные нарушения, но полагается на CLR в плане принудительного применения этого правила во время выполнения. Хотя ссылку на массив типа `Derived[]` можно неявно преобразовать в ссылку типа `Base[]`, любая попытка установить элемент массива способом, не совместимым с системой типов, приведет к исключению `ArrayTypeMismatchException`. Таким образом, в листинге 6.20 это исключение выдается, когда он пытается поместить ссылку на `Base` в массив `Derived[]`.

Безопасность типов сохраняется, и, что удобно, если мы напишем метод, который принимает массив и только читает из него, мы можем передавать в него массивы с производным типом элементов, и это будет работать. Недостатком является то, что CLR для гарантии отсутствия несоответствия типов должен выполнять дополнительную работу во время выполнения, когда вы

изменяете элементы массива. Возможно, он сможет оптимизировать код, чтобы избежать необходимости проверять каждое отдельное присвоение, но какие-то издержки все равно останутся, что делает массивы не такими эффективными, какими они могли бы быть.

Эта причудливая договоренность родом из того времени, когда в .NET были formalizованы понятия ковариантности и контравариантности — они появились вместе с обобщениями, представленными в .NET 2.0. Возможно, если бы обобщения были с самого начала, массивы были бы менее странными. Но стоит сказать, что даже после .NET 2.0 их своеобразная форма ковариантности в течение многих лет была единственным механизмом, встроенным в библиотеку классов, который обеспечивал способ ковариантной передачи коллекции в метод, который хотел получать из нее данные через индексацию. Пока с .NET 4.5 не появился `IReadOnlyList<T>` (для которого `T` является ковариантным), в платформе не было интерфейса индексируемой коллекции только для чтения и, следовательно, стандартного интерфейса индексированной коллекции с параметром ковариантного типа. (`IList<T>`, как и `ICollection<T>`, доступен на чтение/запись, так что он не может быть вариантым.)

Пока мы говорим о совместимости типов и неявных преобразованиях ссылок, которые делают наследование возможным, следует упомянуть еще один тип: `object`.

System.Object

Тип `System.Object`, или `object`, как мы обычно называем его в C#, полезен, так как выступает своего рода обобщенным контейнером: переменная этого типа может содержать ссылку почти на что угодно. Я уже упомянул об этом, но не объяснил, почему это так. Причина в том, что почти все наследуется от `object`.

Если вы не указали базовый класс при написании класса, компилятор C# автоматически использует в качестве него `object`. Как мы скоро увидим, он выбирает другие базовые типы для определенных типов, таких как структуры, но даже они косвенно происходят из `object`. (Как и всегда, типы указателей являются исключением — они не являются производными от `object`.)

Отношения между интерфейсами и объектами более тонкие. Интерфейсы не являются производными от `object`, потому что интерфейс может ука-

зывать в качестве базовых только другие интерфейсы. Однако ссылка на любой тип интерфейса неявно преобразуется в ссылку на тип `object`. Это преобразование всегда допустимо, потому что все типы, которые способны реализовывать интерфейсы, в конечном итоге являются производными от `object`. Более того, C# делает члены класса `object` доступными через ссылки на интерфейсы, хотя они, строго говоря, не являются членами интерфейса. Это означает, что ссылки любого вида всегда содержат следующие методы, определенные `object`: `ToString`, `Equals`, `GetHashCode` и `GetType`.

Вездесущие методы `System.Object`

Я уже использовал `ToString` во многих примерах. Реализация по умолчанию возвращает имя типа объекта, но многие типы предоставляют собственную реализацию `ToString`, возвращая более полезное текстовое представление текущего значения объекта. Например, числовые типы возвращают десятичное представление своего значения, а `bool` возвращает либо `True`, либо `False`.

Я уже говорил об `Equals` и `GetHashCode` в главе 3, но здесь я позволю себе краткое резюме. `Equals` позволяет сравнивать объект с любым другим объектом. Реализация по умолчанию просто выполняет сравнение идентификаторов, т. е. возвращает `true` только тогда, когда объект сравнивается с самим собой. Многие типы предоставляют метод `Equals`, который выполняет сравнение по значению. Например, два различных объекта `string` могут содержать идентичный текст, и в этом случае они сообщат, что равны друг другу. (Если нужно выполнить сравнение объектов на основе идентификаторов, тогда как они проводят сравнение на основе значений, используйте статический метод `ReferenceEquals` класса `object`.) Кстати, `object` также определяет статическую версию `Equals`, которая принимает два аргумента. Метод проверяет, равны ли аргументы `null`, и возвращает `true`, если оба равны `null`, и `false`, если `null` равен только один. В противном случае он полагается на метод `Equals` первого аргумента. И как обсуждалось в главе 3, `GetHashCode` возвращает целое число, которое является сокращенным представлением значения объекта. Оно используется механизмами на основе хеша, такими как класс коллекции `Dictionary< TKey, TValue >`. Любая пара объектов, для которых `Equals` возвращает `true`, должна возвращать одинаковые хеш-коды.

Метод `GetType` дает способ узнать о типе объекта и возвращает ссылку типа `Type`. Это часть API отражения, являющейся темой главы 13.

Помимо указанных открытых членов, доступных через любую ссылку, `object` определяет еще два члена, которые не являются публичными. Только сам объект имеет доступ к этим элементам. Это методы `Finalize` и `MemberwiseClone`. CLR вызывает метод `Finalize`, чтобы уведомить вас о том, что ваш объект больше не используется и занимаемая им память готовится к освобождению. В C# мы обычно не работаем с методом `Finalize` напрямую, потому что в C# этот механизм реализован через деструкторы, что я покажу в главе 7. `MemberwiseClone` создает новый экземпляр того же типа, что и ваш объект, и инициализирует его копиями всех полей вашего объекта. Если вам нужен клон объекта, это может оказаться проще, чем написание кода, который копирует все содержимое вручную, хотя этот способ не так быстр.

Причина, по которой эти два последних метода доступны только изнутри объекта, заключается в том, что вы, скорее всего, не захотите, чтобы другие люди клонировали ваш объект. Кроме того, нет никакой пользы в том, что внешний код сможет вызывать метод `Finalize`, вводя ваш объект в заблуждение сообщением, что он вскоре должен быть освобожден, если на самом деле это не так. Класс `object` ограничивает видимость этих членов, но не делает их `private`, так как это будет означать, что только сам класс `object` может получить к ним доступ, потому что члены с доступом `private` не видны даже производным классам. Вместо этого `object` использует `protected`, спецификатор видимости, разработанный для сценариев наследования.

Доступность и наследование

На этот момент вы уже знакомы с большинством вариантов доступности, доступных для типов и их членов. Элементы, помеченные как `public`, доступны всем, доступ к `private` возможен только из типа, который их объявил, а к `internal` — из кода, определенного в том же компоненте. Но с наследованием мы получаем три дополнительных варианта доступности⁴.

Член, помеченный как `protected`, доступен внутри типа, который его определил, а также внутри любых производных типов. Но для кода, использующего экземпляр вашего типа, члены `protected` недоступны, как и `private`.

Следующий уровень защиты для членов типа — это `protected internal`. (Вы можете написать `internal protected`, так как порядок здесь не имеет

⁴ Точнее, в той же сборке, а также в дружественных сборках. Сборки описываются в главе 12.

значения.) Это делает элемент более доступным, чем `protected` или `internal` по отдельности: элемент будет доступен для всех производных типов и для всего кода в сборке.

Третий уровень защиты, добавляемый наследованием, — это `protected private`. Элементы, помеченные таким образом (или эквивалентным `private protected`), доступны только для типов, которые являются производными и определены в том же компоненте, что и определяющий тип.

Можно использовать `protected`, `protected internal` или `protected private` для любого члена типа, а не только для методов. Применение этих спецификаторов доступа возможно и со вложенными типами.

Хотя члены `protected` (и `protected internal`, но не `protected private`) не доступны через обычную переменную определяющего типа, они все еще являются частью открытого API типа в том смысле, что любой, кто имеет доступ к вашим классам, сможет использовать эти члены. Как и в большинстве языков, которые поддерживают подобный механизм, `protected`-члены в C# обычно используются для предоставления функций, которые производные классы могут посчитать полезными. Если вы напишите класс с доступностью `public`, который поддерживает наследование, то любой может наследовать от него и получить доступ к его членам с доступом `protected`. Поэтому удаление или изменение членов с доступом `protected` может привести к нарушению в работе кода, который зависит от вашего класса, так же как к этому приведет удаление или изменение членов с доступом `public`.



Это ограничение не распространяется на интерфейсы, которые вы реализуете. Класс с доступом `public` может реализовать `public` или `private` интерфейсы. Однако это относится к базовым интерфейсам интерфейса: интерфейс с доступом `public` не может быть производным от интерфейса `internal`.

Когда вы наследуете класс, то сделать свой класс более доступным, чем его базовый класс, нельзя. Например, если вы наследуете от класса `internal`, то не можете объявить свой класс `public`. Ваш базовый класс является частью API вашего класса, поэтому любой, кто хочет использовать ваш класс, также будет использовать его базовый класс. Это означает, что если базовый класс недоступен, ваш класс также будет недоступен. Именно поэтому C#

не позволяет классу быть более доступным, чем его базовый класс. Например, если вы наследуете от вложенного класса с доступом `protected`, ваш производный класс может быть `protected`, `private` или `protected private`, но не `public`, `internal` или `protected internal`.

При определении методов есть еще одно ключевое слово, которое вы можете добавить в интересах производного типа: `virtual`.

Виртуальные методы

Виртуальный метод — это такой метод, который производный тип может заменить. Несколько методов, определенных `object`, являются виртуальными: методы `ToString`, `Equals`, `GetHashCode` и `Finalize` допускают замену. Код, необходимый для создания полезного текстового представления значения объекта, будет значительно отличаться от одного типа к другому, как и логика, необходимая для определения равенства и создания хеш-кода. Типы обычно определяют метод завершения только в том случае, если им нужно выполнить некоторую специализированную очистку, когда они выходят из употребления.

Не все методы являются виртуальными. По сути, по умолчанию C# делает методы невиртуальными. Метод `GetType` класса `object` не является виртуальным, поэтому вы всегда можете доверять информации, которую он возвращает. Вы знаете, что вызываете метод `GetType`, определяемый .NET, а не какую-то зависимую от типа замену, предназначенную для того, чтобы ввести вас в заблуждение. Чтобы объявить виртуальный метод, используйте ключевое слово `virtual`, как показано в листинге 6.22.

В синтаксисе вызова виртуального метода нет ничего необычного. Как показано в листинге 6.23, он выглядит так же, как и вызов любого другого метода.

Листинг 6.22. Класс с виртуальным методом

```
public class BaseWithVirtual
{
    public virtual void ShowMessage()
    {
        Console.WriteLine("Hello from BaseWithVirtual");
    }
}
```



Вы также можете применять ключевое слово `virtual` к свойствам. Свойства — это просто скрытые методы, поэтому по сути вы делаете виртуальными методы чтения свойств. То же самое относится и к событиям, которые обсуждаются в главе 9.

Листинг 6.23. Использование виртуального метода

```
public static void CallVirtualMethod(BaseWithVirtual o)
{
    o.ShowMessage();
}
```

Разница между вызовами виртуальных и не виртуальных методов заключается в том, что вызов виртуального метода во время выполнения решает, какой именно метод вызывать. Код в листинге 6.23 фактически проверяет переданный объект, и если тип объекта предоставляет собственную реализацию `ShowMessage`, он будет вызывать ее вместо той, которая определена в `BaseWithVirtual`. Метод выбирается на основе фактического типа, который целевой объект имеет во время выполнения, а не статического типа выражения (определенного во время компиляции), которое ссылается на целевой объект.



Поскольку вызов виртуального метода выбирает метод в зависимости от типа объекта, для которого метод вызывается, статические методы не могут быть виртуальными.

Производные типы не обязаны заменять виртуальные методы. В листинге 6.24 показаны два класса, производные от класса из листинга 6.22. Первый оставляет реализацию `ShowMessage` базового класса нетронутой. Второй его переопределяет. Обратите внимание на ключевое слово `override` — C# требует от нас явного заявления о том, что мы намерены переопределить виртуальный метод.

Листинг 6.24. Переопределение виртуальных методов

```
public class DeriveWithoutOverride : BaseWithVirtual
{
}

public class DeriveAndOverride : BaseWithVirtual
{
```

```
public override void ShowMessage()
{
    Console.WriteLine("This is an override");
}
}
```

Можно использовать эти типы с методом из листинга 6.23. Листинг 6.25 вызывает его три раза, каждый раз передавая объект другого типа.

Листинг 6.25. Использование виртуальных методов

```
CallVirtualMethod(new BaseWithVirtual());
CallVirtualMethod(new DeriveWithoutOverride());
CallVirtualMethod(new DeriveAndOverride());
```

Это приводит к следующему выводу:

```
Hello from BaseWithVirtual
Hello from BaseWithVirtual
This is an override
```

Очевидно, что, когда передается экземпляр базового класса, мы получаем выходные данные из метода `ShowMessage` базового класса, а также от производного класса, который не предоставил переопределение. И лишь последний класс, который переопределяет метод, производит другой вывод. Это показывает, что виртуальные методы обеспечивают способ написания полиморфного кода: в листинге 6.23 могут использоваться различные типы. Возможно, вы гадаете, зачем это нужно, учитывая, что интерфейсы тоже поддерживают полиморфный код. До C# 8.0 одним из основных преимуществ виртуальных методов перед интерфейсами было то, что базовый класс мог обеспечивать реализацию, которую производные классы будут получать по умолчанию, предоставляя свою собственную реализацию, только если им действительно нужно что-то другое. Добавление к языку реализаций интерфейса по умолчанию означает, что интерфейсы теперь могут делать то же самое, хотя реализация члена интерфейса по умолчанию не может определять или получать доступ к нестатическим полям. Поэтому она несколько ограничена по сравнению с классом, который определяет виртуальную функцию. (А поскольку реализации интерфейса по умолчанию требуют поддержки среды выполнения старше, чем .NET Core 3.0, что включает в себя любую библиотеку, ориентированную на .NET Standard 2.0 или старше.) Виртуальные методы обладают еще одним, менее очевидным преимуществом, но прежде, чем об-

ратиться к нему, стоит изучить функционал виртуальных методов, который на первый взгляд еще больше напоминает работу интерфейсов.

Абстрактные методы

Вы можете определить виртуальный метод, не предоставляя реализацию по умолчанию. С# называет это абстрактным методом. Если класс содержит один или несколько абстрактных методов, то он является неполным, поскольку не реализует все методы, которые определяет. Классы такого типа также называются абстрактными, и создать экземпляры абстрактного класса невозможно; попытка использовать оператор `new` с таким классом приведет к ошибке компилятора. Иногда, когда речь идет о классах, бывает полезно пояснить, что определенный класс не является абстрактным, и для этого мы обычно используем термин «конкретный класс».

Если вы наследуете от абстрактного класса и не предоставляете реализации для всех абстрактных методов, ваш производный класс также будет абстрактным. Следует заявить о своем намерении написать абстрактный класс ключевым словом `abstract`; если оно отсутствует в классе, который имеет нереализованные абстрактные методы (которые определил или унаследовал от своего базового класса), компилятор С# сообщит об ошибке. Листинг 6.26 показывает абстрактный класс, который определяет один абстрактный метод. Абстрактные методы являются виртуальными по определению; не было бы большого смысла в определении метода, у которого нет тела, при отсутствии у производных классов возможности это тело предоставить.

Объявления абстрактных методов просто определяют сигнатуру, но не содержат тела. В отличие от интерфейсов, каждый абстрактный член имеет свою собственную доступность — вы можете объявить абстрактные методы `public`, `internal`, `protected internal`, `protected private` или `protected`. (Нет смысла назначать абстрактному или виртуальному методу доступ `private`, потому что метод окажется недоступен для производных типов и, следовательно, его невозможно переопределить.)

Листинг 6.26. Абстрактный класс

```
public abstract class AbstractBase
{
    public abstract void ShowMessage();
}
```



Хотя классы, содержащие абстрактные методы, обязаны быть абстрактными, обратное уже неверно. Законно, хотя и необычно, определять класс абстрактным, даже если это вполне жизнеспособный конкретный класс. Это предотвращает создание класса. Производный класс будет конкретным классом без необходимости переопределения каких-либо абстрактных методов.

Абстрактные классы могут объявлять, что реализуют интерфейс, но не предоставлять полной реализации. Вы не можете просто опустить нереализованные члены. Вы должны явно объявить все его члены, пометив те, которые вы хотите оставить без реализации как абстрактные, как это показано в листинге 6.27. Это заставляет конкретные производные типы предоставлять реализацию.

Листинг 6.27. Реализация абстрактного интерфейса

```
public abstract class MustBeComparable : IComparable<string>
{
    public abstract int CompareTo(string other);
}
```

Очевидно, что есть некоторые пересечения между абстрактными классами и интерфейсами. Оба предоставляют способ определения абстрактного типа, который может использоваться кодом без необходимости знания точного типа, который будет предоставлен во время выполнения. У каждого варианта есть свои плюсы и минусы. Преимущество интерфейсов состоит в том, что один тип может реализовывать несколько интерфейсов, тогда как класс может указывать только один базовый класс. Но абстрактные классы могут определять поля и использовать их в любых своих реализациях элементов по умолчанию, а также они предоставляют способ написания реализаций по умолчанию, которые будут работать в средах выполнения более старых, чем .NET Core 3.0. Тем не менее есть и менее заметное преимущество виртуальных методов, которое вступает в дело, когда вы со временем выпускаете несколько версий библиотеки.

Наследование и управление версиями библиотеки

Представьте, что произойдет, если вы написали и выпустили библиотеку, в которой определены некоторые открытые интерфейсы и абстрактные классы, а во втором выпуске библиотеки решили, что хотите добавить не-

сколько новых членов к одному из интерфейсов. Может так получиться, что у клиентов, использующих ваш код, не возникнет никаких проблем.

Конечно, любого места, где они используют ссылку этого типа интерфейса, некоснется добавление нового функционала. Но что, если некоторые из клиентов имеют написанные типы, которые реализуют ваш интерфейс? Предположим, например, что в будущей версии .NET Microsoft решила добавить новый член в интерфейс `IEnumerable<T>`.

До C# 8.0 это было бы катастрофой. Этот интерфейс не только широко используется, но также множество раз реализован. Классы, которые уже реализуют `IEnumerable<T>`, перестанут работать, потому что не смогут предоставить этот новый член. Поэтому старый код не будет компилироваться, а уже скомпилированный код приведет к ошибкам `MissingMethodException` во время выполнения. Введение в C# 8.0 поддержки в интерфейсах реализаций членов по умолчанию сглаживает эту проблему: в маловероятном случае добавления компанией Microsoft нового члена в `IEnumerable<T>`, он может предоставить реализацию по умолчанию, предотвращающую указанные ошибки. Однако есть и более изощренная проблема. Некоторые классы могут по совпадению иметь элемент с тем же именем и сигнатурой, что и у недавно добавленного метода. Если этот код перекомпилируется с новым определением интерфейса, компилятор будет рассматривать этот существующий элемент как часть реализации интерфейса, даже если разработчик, который написал метод, не писал его с таким намерением. Таким образом, если существующий код по стечению обстоятельств не выполняет того, что требует новый член, возникнет проблема, но мы не увидим ошибок компилятора или предупреждений.

В результате широко распространенное правило гласит, что не следует изменять интерфейсы после их публикации. Если у вас есть полный контроль над всем кодом, который использует и реализует интерфейс, изменения интерфейса могут сойти вам с рук, так как вы можете внести любые необходимые изменения в уязвимый код. Но как только интерфейс стал доступен для использования в базах исходного кода, которыми вы не управляете, т. е. после его публикации, его уже невозможно изменить, не рискуя поломать чужой код. Реализации интерфейса по умолчанию снижают этот риск, но они не способны устранить проблему, связанную с тем, что существующие методы случайно неверно интерпретируются при их повторной компиляции с обновленным интерфейсом.

Абстрактные базовые классы не должны страдать от этой проблемы. Очевидно, что введение новых абстрактных членов приведет к точно таким же ошибкам `MissingMethodException`, но введение новых виртуальных методов — нет. (И поскольку виртуальные методы были в C#, начиная с v1, это дает возможность ориентироваться на среды выполнения старше, чем .NET Core 3.0, где поддержка реализации интерфейса по умолчанию недоступна.)

Но что, если после выпуска версии 1.0 компонента вы добавите в v1.1 новый виртуальный метод, который, как оказалось, будет иметь то же имя и сигнатуру, что и метод, который один из ваших клиентов добавил в производный класс? Возможно, в версии 1.0 ваш компонент определяет довольно неинтересный базовый класс, показанный в листинге 6.28.

Листинг 6.28. Базовый тип версии 1.0

```
public class LibraryBase
{
}
```

Если вы выпустите эту библиотеку, скажем, как отдельный продукт или как часть какого-либо SDK для своего приложения, клиент вполне может написать производный тип, такой как в листинге 6.29. Написанный им метод `Start` явно не предназначен для переопределения чего-либо в базовом классе.

Листинг 6.29. Класс, наследуемый от базового версии 1.0

```
public class CustomerDerived : LibraryBase
{
    public void Start()
    {
        Console.WriteLine("Derived type's Start method");
    }
}
```

Поскольку вы не обязательно видите каждую строку кода ваших клиентов, то можете не знать о методе `Start`. Соответственно, вы можете решить добавить новый виртуальный метод в версию 1.1 вашего компонента, также названный `Start`, как показано в листинге 6.30.

Листинг 6.30. Базовый тип версии 1.1

```
public class LibraryBase
{
    public virtual void Start() { }
}
```

Представьте, что система вызывает этот метод как часть процедуры инициализации, представленной в v1.1. Вы определили пустую реализацию по умолчанию, так что типам, производным от `LibraryBase`, которым нет необходимости участвовать в этой процедуре, не нужно ничего делать. Типы же, которые хотят участвовать в инициализации, переопределят этот метод. Но что произойдет с классом из листинга 6.29? Очевидно, что разработчик, который его написал, не собирался участвовать в вашем новом механизме инициализации, потому что на момент написания кода его еще не было. Может случиться что-то плохое, если ваш код решит вызвать метод `Start` класса `CustomerDerived`, так как разработчик предположительно ожидает, что его метод будет вызываться только тогда, когда его собственный код решит его вызвать. К счастью, компилятор обнаружит эту проблему. Если клиент пытается скомпилировать листинг 6.29 с версией 1.1 вашей библиотеки (листинг 6.30), компилятор предупредит его, что что-то не так:

```
warning CS0114: 'CustomerDerived.Start()' hides inherited member  
'LibraryBase.Start()'. To make the current member override that  
implementation, add the override keyword. Otherwise add the new  
keyword.5
```

Вот почему компилятору C# требуется ключевое слово `override`, когда мы заменяем виртуальные методы. Он хочет знать, намеревались ли мы переопределить существующий метод, чтобы в противном случае он мог предупредить нас о коллизиях. (Отсутствие какого-либо эквивалентного ключевого слова, обозначающего намерение реализовать элемент интерфейса, — причина того, что компилятор не может обнаружить ту же проблему с реализацией интерфейса по умолчанию. А отсутствует оно потому, что реализации интерфейса по умолчанию до C# 8.0 не существовало.)

Мы получаем предупреждение вместо ошибки, потому что компилятор обеспечивает поведение, которое, скорее всего, будет безопасным на момент, когда эта ситуация возникла из-за выпуска новой версии библиотеки. Компилятор догадывается — в данном случае верно, — что разработчик, написавший тип `CustomerDerived`, не хотел переопределять метод `Start` класса `LibraryBase`. Поэтому вместо того, чтобы метод `Start` типа `CustomerDerived` переопределял виртуальный метод базового класса, он скрывает его. Говорят, что производный тип скрывает член базового класса, когда определяет новый член с тем же именем.

⁵ Чтобы текущий член переопределил эту реализацию, добавьте ключевое слово `override`. В противном случае добавьте ключевое слово `new`.

Сокрытие методов — совсем не то же самое, что их переопределение. Когда происходит сокрытие, базовый метод не заменяется. Листинг 6.31 показывает, что скрытый метод `Start` остается доступным. Он создает объект `CustomerDerived` и помещает ссылку на этот объект в две переменные разных типов: одну типа `CustomerDerived` и одну типа `LibraryBase`. Затем через каждую из них он вызывает `Start`.

Листинг 6.31. Скрытый метод против виртуального

```
var d = new CustomerDerived();
LibraryBase b = d;

d.Start();
b.Start();
```

Когда мы используем переменную `d`, вызов `Start` в конечном итоге вызывает метод `Start` производного типа, который скрыл член базового класса. Но тип переменной `b` — `LibraryBase`, так что вызывается метод `Start` базового класса. Если бы `CustomerDerived` переопределял метод `Start` базового класса, а не скрывал его, оба этих вызова указывали бы на переопределенный метод.

Когда происходят конфликты имен из-за новой версии библиотеки, такое поведение сокрытия обычно является правильным. Если в коде клиента есть переменная типа `CustomerDerived`, то этот код захочет вызвать метод `Start`, специфичный для этого производного типа. Тем не менее компилятор выдает предупреждение, потому что не знает наверняка, является ли это причиной проблемы. Возможно, вы *действительно* хотели переопределить метод и просто забыли написать ключевое слово `override`.

Как и большинству разработчиков, мне не нравятся предупреждения компилятора, и я стараюсь избегать написания кода, который их производит. Но что делать, если новая версия библиотеки поставит вас в такую ситуацию? Очевидно, что лучшее долгосрочное решение состоит в изменении метода в вашем производном классе, чтобы он не конфликтовал с методом в новой версии библиотеки. Но если поджимают сроки, может потребоваться более подходящее решение. Так что C# позволяет объявить, что вы знаете о конфликте имен и определенно хотите скрыть элемент базового класса, а не переопределить его. Как показано в листинге 6.32, можно использовать ключевое слово `new`, чтобы показать, что вы знаете о проблеме и точно хотите скрыть член базового класса. Код будет по-прежнему вести себя так же, но вы больше не увидите предупреждений, так как заверили компилятор в том, что в курсе происходящего. Но эту проблему все равно придется решать, так как

рано или поздно существование в одном типе двух методов с одинаковыми именами, которые значат разное, вызовет путаницу.

Листинг 6.32. Избегание предупреждений при скрытии членов

```
public class CustomerDerived : LibraryBase
{
    public new void Start()
    {
        Console.WriteLine("Derived type's Start method");
    }
}
```



C# не позволяет использовать ключевое слово `new` для решения такой же проблемы, возникающей при реализации интерфейса по умолчанию. Невозможно сохранить реализацию по умолчанию, предоставляемую интерфейсом, и при этом объявить открытый метод с той же сигнатурой. Это расстраивает, поскольку возможно на двоичном уровне: такое поведение вы получаете, если не перекомпилируете код, реализующий интерфейс после добавления нового члена с реализацией по умолчанию. У вас все еще могут быть отдельные реализации, скажем `ILibrary.Start` и `CustomerDerived.Start`, но придется использовать явную реализацию интерфейса.

Время от времени вам будет встречаться ключевое слово `new`, используемое таким образом, по причинам, не связанным с решением проблемы управления версиями библиотеки. Например, интерфейс `ISet<T>`, который я показал в главе 5, использует его для добавления нового метода `Add`. `ISet<T>` наследуется от интерфейса `ICollection <T>`, который уже содержит метод `Add`, принимающий экземпляр `T` и возвращающий тип `void`. `ISet<T>` вносит небольшое изменение, показанное в листинге 6.33.

Листинг 6.33. Скрытие для изменения сигнатуры

```
public interface ISet<T> : ICollection<T>
{
    new bool Add(T item);
    // ... другие члены пропущены для ясности
}
```

Метод `Add` интерфейса `ISet<T>` сообщает вам, был ли элемент, который вы только что добавили, уже в наборе. Это то, что не поддерживается методом `Add` базового интерфейса `ICollection<T>`. `ISet<T>` нужно, чтобы его `Add` имел

другой тип возвращаемого значения — `bool` вместо `void`, — поэтому он определяет `Add` с ключевым словом `new`, чтобы указать, что тот должен скрывать аналогичный метод в `ICollection<T>`. Оба метода все еще доступны — если у вас есть две ссылающиеся на один и тот же объект переменные, одна из которых типа `ICollection<T>`, а другая — `ISet<T>`, вы можете получить доступ к `Add` с `void` через первую, а к `Add` с `bool` — через вторую.

Microsoft не должна была этого делать. Можно было бы вызывать новый метод `Add` как-нибудь еще — например, `AddIfNotPresent`. Но, несомненно, менее запутанный способ — это иметь единое имя метода для добавления элементов в коллекцию, особенно если учесть, что вы можете игнорировать возвращаемое значение, и в такой момент новый `Add` выглядит неотличимым от старого. И большинство реализаций `ISet<T>` будут реализовывать метод `ICollection<T>.Add`, напрямую вызывая метод `ISet<T>.Add`, поэтому вполне разумно, чтобы они имели одинаковые имена.

Если отвлечься от предыдущего примера, до сих пор я обсуждал скрытие метода только в контексте компиляции старого кода с новой версией библиотеки. Но что произойдет, если у вас есть старый код, скомпилированный со старой библиотекой, но в итоге ему приходится работать с новой? Это сценарий, с которым вы, скорее всего, столкнетесь, когда рассматриваемая библиотека является библиотекой классов .NET. Предположим, вы используете сторонние компоненты, которые у вас есть только в двоичном виде (например, те, которые вы купили у компании, которая не предоставляет исходный код). Провайдер собирает их для использования с определенной версией .NET. Если вы обновляете свое приложение для запуска с новой версией .NET, вы можете не иметь доступа к более свежим версиям сторонних компонентов из-за того, что провайдер еще не выпустил их или, возможно, прекратил деятельность.

Если используемые вами компоненты были скомпилированы, скажем, для .NET Standard 1.2 и вы используете их в проекте, созданном для .NET Core 3.0, то все эти старые компоненты в конечном итоге будут использовать версии библиотеки классов .NET Core 3.0. В .NET существует версионная политика, в соответствии с которой все используемые конкретной программой компоненты используют одну и ту же версию библиотеки классов независимо от того, для какой версии мог быть создан отдельный компонент. Поэтому вполне возможно, что какой-то компонент, `OldControls.dll`, содержит классы, которые являются производными от классов в .NET Standard 1.2

и определяют члены, которые вступают в противоречие с именами членов, добавленных в недавнюю .NET Core 3.0.

Это более или менее тот сценарий, который я описал ранее, за исключением того, что код, написанный для более старой версии библиотеки, перекомпилироваться не будет. Мы не получим предупреждение компилятора о скрытии метода, потому что это потребует запуска компилятора, а у нас есть только двоичный файл соответствующего компонента. Что же делать в этом случае?

К счастью, перекомпилировать старый компонент не нужно. Компилятор C# устанавливает различные флаги в выводе для каждого компилируемого метода, указывая, является ли метод виртуальным и был ли он предназначен для переопределения какого-либо метода в базовом классе. Когда вы помещаете в определение метода ключевое слово `new`, компилятор устанавливает флаг, указывающий, что метод не предназначен для переопределения чего-либо. CLR называет это флагом `newslot`. Когда C# компилирует метод, подобный методу из листинга 6.29, который не содержит ни `override`, ни `new`, он также устанавливает для этого метода флаг `newslot`, потому что во время компиляции метода в базовом классе не было метода с тем же именем. Как с точки зрения разработчика, так и с точки зрения компилятора метод `Start` класса `CustomerDerived` был написан как совершенно новый метод, не связанный ни с чем в базовом классе.

Поэтому, когда этот старый компонент загружается вместе с новой версией библиотеки, определяющей базовый класс, CLR видит первоначальную задумку: с точки зрения автора класса `CustomerDerived`, метод `Start` не должен был что-либо переопределять. Поэтому он рассматривает метод `CustomerDerived.Start` как отдельный от `LibraryBase.Start` и скрывает базовый метод, как тогда, когда мы смогли его перекомпилировать.

Кстати, все, что я сказал о виртуальных методах, может также применяться к свойствам, потому что методы доступа к свойствам — это обычные методы. Таким образом, вы можете определять виртуальные свойства, а производные классы могут переопределять или скрывать их точно так же, как методы. Я не буду обращаться к событиям до главы 9, но события — это тоже замаскированные методы, следовательно, они также могут быть виртуальными.

Иногда вам может понадобиться написать класс, который переопределяет виртуальный метод, а затем предотвращает его переопределение производными классами. Для этого C# определяет ключевое слово `sealed`, и на самом деле запечатывать можно не только методы.

Запечатанные методы и классы

Виртуальные методы преднамеренно открыты для модификации через наследование. Запечатанный метод — это метод, который, наоборот, нельзя переопределить. В C# методы запечатаны по умолчанию: они не могут быть переопределены, если не объявлены виртуальными. Но когда вы переопределяете виртуальный метод, вы можете запечатать его, закрыв для дальнейшей модификации. В листинге 6.34 этот подход используется для предоставления пользовательской реализации `ToString`, которую нельзя переопределить производными классами.

Листинг 6.34. Запечатанный метод

```
public class FixedToString
{
    public sealed override string ToString()
    {
        return "Arf arf!";
    }
}
```

Вы также можете запечатать целый класс, запретив наследовать от него. В листинге 6.35 показан класс, который не только ничего не делает, но и не позволяет расширять его, чтобы он начал делать что-то полезное. (Обычно запечатывается только класс, который хоть что-то делает. Этот пример просто показывает, где размещать ключевое слово.)

Листинг 6.35. Запечатанный класс

```
public sealed class EndOfTheLine
{}
```

Некоторые типы изначально запечатаны. Например, значимые типы не поддерживают наследование, поэтому структуры и перечисления по сути запечатаны. Встроенный класс `string` также запечатан.

Есть две разумные причины для запечатывания классов или методов. Первая состоит в том, что если вы хотите гарантировать какой-то конкретный инвариант, то, оставив свой тип открытым для модификации, вы не сможете этого сделать. Например, экземпляры типа `string` являются неизменяемыми. Сам по себе тип `string` не предоставляет способа изменить значение экземпляра, и, поскольку никто не может получить производный класс от `string`,

вы можете гарантировать, что если у вас есть ссылка на тип `string`, то это ссылка на неизменяемый объект. Это делает безопасным его использование в сценариях, в которых вы не хотите, чтобы его значение изменялось. Например, когда вы используете объект в качестве ключа к словарю (или чему-то еще, что зависит от хеш-кода), то значение не должно изменяться, потому что если хеш-код изменяется, когда элемент используется в качестве ключа, контейнер будет работать неправильно.

Другая причина оставлять что-либо запечатанным состоит в том, что разработка типов, которые могут быть успешно изменены с помощью наследования, — трудоемкая задача, особенно если ваш тип будет использоваться за пределами компании. Простого открытия вещей для модификации недостаточно — если вы решите сделать все ваши методы виртуальными, людям, использующим ваш тип, будет проще изменить его поведение, но вы копаете себе яму, если речь идет о поддержании работоспособности базового класса. Если вы не контролируете весь код, наследуемый от вашего класса, то практически невозможно что-либо изменить в базовом классе — вы никогда не узнаете, какие методы могли быть переопределены в производных классах, что затруднит обеспечение непротиворечивости внутреннего состояния вашего класса. Разработчики, пишущие производные типы, несомненно, сделают все возможное, чтобы ничего не ломать, но они неизбежно будут полагаться на те аспекты поведения вашего класса, которые не были задокументированы. Таким образом, открывая все части вашего класса для модификации через наследование, вы лишаете себя свободы изменять свой класс.

Следует очень внимательно относиться к тому, какие методы вы делаете виртуальными. Дополнительно вносите в документацию информацию о том, разрешено ли вызывающей стороне полностью заменять метод или же она должна вызывать базовую реализацию как часть своего переопределения. Кстати, а как это делается?

Доступ к членам базового класса

Все, что находится в области доступности в базовом классе и не является закрытым, также будет в области доступности и доступа в производном типе. Если требуется получить доступ к какому-либо члену базового класса, то обратитесь к нему, как если бы он был обычным членом вашего класса. Можно получить доступ к членам через ссылку `this` или просто обратиться к ним по имени без квалификации.

Но в некоторых ситуациях нужно указать, что вы явно обращаетесь к члену базового класса. В частности, если вы переопределили метод, то вызов этого метода по имени приведет к рекурсивному вызову переопределения. Если вы хотите вызвать исходный переопределенный метод, то для этого есть специальное ключевое слово, показанное в листинге 6.36.

Листинг 6.36. Вызов базового метода после переопределения

```
public class CustomerDerived : LibraryBase
{
    public override void Start()
    {
        Console.WriteLine("Derived type's Start method");
        base.Start();
    }
}
```

Используя ключевое слово `base`, мы отказываемся от обычного механизма диспетчеризации виртуальных методов. Если бы мы просто написали `Start()`, это привело бы к рекурсивному вызову, который в данном случае был бы нежелателен. Написав `base.Start()`, мы получаем метод, который был бы доступен для экземпляра базового класса, т. е. который мы переопределили.

В этом примере я вызвал реализацию базового класса после завершения своей работы. С# не волнует, когда вы обращаетесь к `base` — это можно делать в начале, в конце или в середине метода. К нему даже можно обратиться несколько раз или не обращаться вообще. Автор базового класса должен задокументировать, должна ли реализация метода базового класса вызываться переопределением и в какой момент.

Используйте ключевое слово `base` и для других членов, таких как свойства и события.

Однако доступ к базовым конструкторам работает немного иначе.

Наследование и конструирование

Хотя производный класс наследует все члены базового, в случае конструктора это означает не то же самое, что в случае всего остального. Другие члены, если они являются публичными в базовом классе, в производном классе останутся публичными для всех, кто его использует. Но конструкторы ведут

себя по-особенному, потому что кто-то, использующий ваш класс, не может создать его экземпляр с помощью одного из конструкторов, определенных базовым классом.

Достаточно очевидно, почему это так: если вам нужен экземпляр некоторого типа D, то вы захотите, чтобы это был полноценный D, в котором все должным образом инициализировано. Предположим, что D унаследован от B. Если бы вы могли использовать один из конструкторов B напрямую, он ничего бы не сделал с частью, специфичной для D. Конструктор базового класса не будет знать ни о каких полях, определенных производным классом, и поэтому не сможет их инициализировать. Если вы хотите D, вам понадобится конструктор, который знает, как инициализировать D. Таким образом, с производным классом вы можете использовать только конструкторы, предлагаемые этим производным классом, независимо от того, какие конструкторы имеются в базовом классе.

В примерах, которые приводились в этой главе, я мог игнорировать это из-за конструктора по умолчанию, который предоставляет C#. Как вы знаете из главы 3, C# добавляет за вас конструктор, который не принимает аргументов, если не написать его самостоятельно. Это делается и для производных классов, и сгенерированный конструктор вызовет конструктор без аргументов базового класса. Но все изменится, когда я начну писать свои собственные конструкторы. Листинг 6.37 определяет пару классов, где base определяет явный конструктор без аргументов, а производный класс определяет тот, который требует один аргумент.

Поскольку базовый класс имеет конструктор с нулевым аргументом, его можно создать с помощью new BaseWithZeroArgCtor(). Но я не могу проделать это с производным типом: его можно построить, только передав аргумент, например с помощью new DerivedNoDefaultCtor(123). Что касается публичного API-интерфейса DerivedNoDefaultCtor, то производный класс, похоже, не унаследовал конструктор своего базового класса.

Листинг 6.37. Отсутствие конструктора по умолчанию в производном классе

```
public class BaseWithZeroArgCtor
{
    public BaseWithZeroArgCtor()
    {
        Console.WriteLine("Base constructor");
    }
}
```

```
public class DerivedNoDefaultCtor : BaseWithZeroArgCtor
{
    public DerivedNoDefaultCtor(int i)
    {
        Console.WriteLine("Derived constructor");
    }
}
```

Но фактически он его унаследовал: посмотрите на вывод, получаемый при создании экземпляра производного типа:

```
Base constructor
Derived constructor
```

При создании экземпляра `DerivedNoDefaultCtor` конструктор базового класса запускается непосредственно перед конструктором производного класса. Поскольку базовый конструктор сработал, то явно никуда не делся. Все конструкторы базового класса доступны для производного типа, но они могут вызываться только конструкторами производного класса. Листинг 6.37 вызывал базовый конструктор неявно: все конструкторы должны вызывать конструктор в своем базовом классе, и, если вы не укажете, какой из них вызывать, компилятор вызовет за вас базовый конструктор с нулевым аргументом.

Что, если базовый тип не определяет конструктор без параметров? В этом случае вы получите ошибку компилятора, если не укажете в производном классе, какой конструктор вызывать. В листинге 6.38 показан базовый класс без конструктора с нулевым аргументом. (Наличие явных конструкторов отключает обычную генерацию компилятором конструктора по умолчанию, и поскольку базовый класс предоставляет только конструктор с аргументами, это будет означать, что конструктора с нулевым аргументом не окажется.) Он также показывает производный класс с двумя конструкторами, оба из которых явно вызывают базовый конструктор, используя ключевое слово `base`.

Листинг 6.38. Явный вызов конструктора базового класса

```
public class BaseNoDefaultCtor
{
    public BaseNoDefaultCtor(int i)
    {
        Console.WriteLine("Base constructor: " + i);
    }
}
```

```
public class DerivedCallingBaseCtor : BaseNoDefaultCtor
{
    public DerivedCallingBaseCtor()
        : base(123)
    {
        Console.WriteLine("Derived constructor (default)");
    }

    public DerivedCallingBaseCtor(int i)
        : base(i)
    {
        Console.WriteLine("Derived constructor: " + i);
    }
}
```

В данном случае производный класс решает предоставить конструктор без параметров даже с учетом того, базовый класс его не имеет. Для этого он передает в качестве его аргумента константное значение. Второй просто передает свой аргумент в конструктор базового класса.



Возникает вопрос: каким образом можно предоставить те же конструкторы, представляющие базовый класс, и те, которые просто передают аргументы дальше? Ответ: пишите все конструкторы вручную. Нет способа заставить C# сгенерировать набор конструкторов в производном классе, которые будут выглядеть идентично предлагаемым базовым классом. Придется идти длинной дорогой.

По крайней мере, Visual Studio может написать этот код за вас — если вы щелкните на объявление класса, а затем на появившийся значок **Quick Actions**, среда предложит сгенерировать конструкторы с теми же аргументами, что и у любого открытого конструктора в базовом классе, автоматически передавая туда все аргументы.

Как было показано в главе 3, инициализаторы полей класса запускаются перед его конструктором. Картина становится более запутанной, если речь идет о наследовании, поскольку в дело вступают несколько классов и несколько конструкторов. Самый простой способ предсказать, что произойдет, — это понять, что, хотя инициализаторы и конструкторы полей экземпляров имеют отдельный синтаксис, C# в конечном итоге компилирует весь код инициализации определенного класса в конструктор. Этот код выполняет следующие шаги: во-первых, он запускает инициализаторы полей, специфичные для этого класса (поэтому этот шаг не включает инициализаторы

полей базового класса, который сам позаботится о себе); затем он вызывает конструктор базового класса; и наконец, выполняет тело конструктора. Как результат в производном классе инициализаторы поля вашего экземпляра будут выполняться до того, как произойдет построение базового класса — не только до выполнения тела конструктора базового класса, но даже до того, как поля экземпляра базового класса окажутся инициализированы. Листинг 6.39 иллюстрирует это.

Листинг 6.39. Изучение порядка выполнения конструктора

```
public class BaseInit
{
    protected static int Init(string message)
    {
        Console.WriteLine(message);
        return 1;
    }

    private int b1 = Init("Base field b1");

    public BaseInit()
    {
        Init("Base constructor");
    }

    private int b2 = Init("Base field b2");
}

public class DerivedInit : BaseInit
{
    private int d1 = Init("Derived field d1");

    public DerivedInit()
    {
        Init("Derived constructor");
    }

    private int d2 = Init("Derived field d2");
}
```

Я поместил инициализаторы полей с обеих сторон конструктора, чтобы показать, что их положение относительно членов, не являющихся полями, не имеет значения. Порядок полей имеет значение, но только по отношению друг к другу. Создание экземпляра класса `DerivedInit` приводит к следующему выводу:

```
Derived field d1
Derived field d2
Base field b1
Base field b2
Base constructor
Derived constructor
```

Он подтверждает, что инициализаторы полей производного типа запускаются первыми, после них инициализаторы базового поля, затем конструктор базового типа и, наконец, конструктор производного. Другими словами, если выполнение тел конструкторов начинается с базового класса, инициализация полей экземпляров происходит в обратном порядке.

Вот почему нельзя вызывать методы экземпляра в инициализаторах полей. Статические методы вызывать можно, но методы экземпляров — нет, потому что класс еще далек от готовности. Может стать проблемой, если один из инициализаторов полей производного типа сумеет вызвать метод базового класса, потому что на тот момент базовый класс еще вообще не выполнял инициализацию — не выполнялось не только его тело конструктора, но и инициализаторы его полей. Если бы на этом этапе были доступны методы экземпляра, нам пришлось бы защищать наш код, потому что нет никаких причин предполагать, что наши поля содержат что-то полезное.

Как видите, тела конструкторов выполняются относительно поздно, поэтому разрешено вызывать из них методы. Но здесь все еще кроется потенциальная опасность. Что, если базовый класс определяет виртуальный метод и вызывает этот метод у себя в своем конструкторе? Если производный тип переопределяет его, мы будем вызывать метод до запуска тела конструктора производного типа. (Однако его инициализаторы полей в этот момент уже сработают. Фактически это основная причина, по которой инициализаторы полей выполняются в обратном порядке — это означает, что у производных классов есть способ выполнить некоторую инициализацию, прежде чем конструктор базового класса сможет вызвать виртуальный метод.) Если вы знакомы с C++, то можете предположить, что когда базовый конструктор вызовет виртуальный метод, он запустит базовую реализацию. Но C# делает это по-другому: в этом случае конструктор базового класса будет вызывать переопределение производного класса. Это не обязательно проблема, и даже может быть иногда полезным, но это также означает, что нужно тщательно продумать и четко документировать свои предположения в случае, если вы хотите, чтобы ваш объект вызывал у себя виртуальные методы во время конструирования.

Специальные базовые типы

Библиотека классов .NET определяет несколько базовых типов, которые имеют особое значение в C#.

Наиболее очевидным является `System.Object`, который я уже подробно описал.

Там также есть и `System.ValueType`. Это абстрактный базовый тип для всех значимых типов, поэтому любая определяемая вами структура, а также все встроенные значимые типы, такие как `int` и `bool`, наследуются из `ValueType`. Забавно, что сам по себе `ValueType` является ссылочным типом и только его производные типы являются значимыми типами. Как и большинство типов, `ValueType` — производный тип от `System.Object`. Здесь есть явная концептуальная трудность: в общем случае производные классы — это все то, чем является их базовый класс, плюс все добавляемые ими функциональные возможности. Таким образом, учитывая, что `object` и `ValueType` являются ссылочными типами, может показаться странным, что типы, производные от `ValueType`, таковыми не являются. И в этом отношении не очевидно, как переменная типа `object` может содержать ссылку на экземпляр чего-то, что не является ссылочным типом. Все эти вопросы я сниму в главе 7.

C# не позволяет вам явно наследовать от `ValueType`. Если нужен тип, производный от `ValueType`, то для этого используется ключевое слово `struct`. Вы можете объявить переменную типа `ValueType`, но так как этот тип не определяет никаких открытых членов, ссылка на `ValueType` не позволяет ничего, что вы не можете сделать со ссылкой на `object`. Единственное заметное отличие состоит в том, что с переменной этого типа вы можете присваивать экземпляры любого значимого типа, но не экземпляры ссылочного типа. За исключением этого, тип идентичен `object`. В результате `ValueType` довольно редко упоминается в коде C# в явном виде.

Типы-перечисления, в свою очередь, все тоже происходят из общего абстрактного базового типа: `System.Enum`. Поскольку перечисления — это значимые типы, вас не должно удивить, что `Enum` — это производный тип от `ValueType`. Как и в случае с `ValueType`, вы бы не сумели наследовать от `Enum` явно, поэтому для этого используется ключевое слово `enum`. В отличие от `ValueType`, `Enum` добавляет несколько полезных членов. Например, его статический метод `GetValues` возвращает массив всех значений перечисления, в то время как `GetNames` возвращает массив со всеми этими значениями, пре-

образованными в строки. Он также определяет `Parse`, который преобразует строковое представление обратно в значение перечисления.

Как описано в главе 5, все массивы происходят от общего базового класса `System.Array`, и вы уже видели определяемые им функции.

Базовый класс `System.Exception` – особенный: когда вы генерируете исключение, C# требует, чтобы объект, который вы генерируете, был либо этого типа, либо производного от него. (Исключения – тема главы 8.)

Все типы делегатов являются производными от общего базового типа `System.MulticastDelegate`, который, в свою очередь, берет начало в `System.Delegate`. О них я расскажу в главе 9.

Это все базовые типы, которые CTS рассматривает как особые. Имеется еще один базовый тип, которому компилятор C# придает особое значение, и это – `System.Attribute`. В главе 1 я добавил кое-какие аннотации к методам и классам, чтобы дать понять инфраструктуре юнит-теста, что они должны обрабатываться особым образом. Все эти атрибуты соответствуют типам, поэтому, когда я применил атрибут `[TestClass]` к классу, то использовал тип `TestClassAttribute`. Все типы, предназначенные для использования в качестве атрибутов, должны быть производными от `System.Attribute`. Некоторые из них распознаются компилятором, например те, которые управляют номерами версий, помещаемых компилятором в заголовки файлов EXE и DLL, которые он создает. Обо всем этом в главе 14.

Итог

C# поддерживает единичное наследование реализации, и то только с классами – от структуры наследовать не получится. Однако интерфейсы могут объявлять несколько базовых интерфейсов, а класс способен реализовывать несколько интерфейсов. Существуют неявные преобразования ссылок из производных типов в базовые типы, и обобщенные интерфейсы, и делегаты могут содержать дополнительные неявные преобразования ссылок с использованием либо ковариантности, либо контравариантности. Все типы являются производными от `System.Object`, что гарантирует для всех переменных доступ к определенным стандартным членам. Мы увидели, как виртуальные методы позволяют производным классам изменять выбранные члены их базовых классов и как запечатывание может отключить этот функционал. Мы также рассмотрели отношения между производным

типов и его базовым типом, когда дело касается, в частности, доступа к членам и конструкторам.

Обзор наследования завершен, и он выявил некоторые новые проблемы — взаимоотношения значимых типов и ссылок, а также роль методов завершения. В следующей главе я расскажу о связи между ссылками и жизненным циклом объекта, а также о том, как CLR преодолевает разрыв между ссылками и значимыми типами.

ГЛАВА 7

Время жизни объекта

Одним из достоинств модели управляемого исполнения .NET является то, что среда выполнения способна автоматизировать большую часть работы по управлению памятью в вашем приложении. Я показал множество примеров создания объектов с помощью ключевого слова `new`, но ни в одном из них явно не освобождал память, занятую этими объектами.

В большинстве случаев не нужно предпринимать каких-либо действий для освобождения памяти. Среда выполнения предоставляет *сборщик мусора* (GC), механизм, который автоматически обнаруживает, когда объекты больше не используются, и восстанавливает память, которую они занимали, чтобы ее можно было использовать для новых объектов¹. Однако существуют определенные схемы использования, которые могут вызвать проблемы с производительностью или даже полностью свести на нет работу сборщика мусора, поэтому полезно понимать, как он работает. Это особенно важно для долго выполняющихся процессов, которые могут работать в течение нескольких дней (процессы с коротким сроком службы вполне могут пережить несколько утечек памяти).

Сборщик мусора разработан для эффективного управления памятью, но память — не единственный ограниченный ресурс, с которым вам, скорее всего, придется иметь дело. Некоторые вещи занимают небольшой объем памяти в CLR, но при этом являются достаточно затратными, например подключение к базе данных или дескриптор из API операционной системы. Сборщик мусора не всегда хорошо с этим справляется, поэтому я объясню работу `IDisposable`, интерфейса, предназначенного для работы с тем, что должно быть освобождено более срочно, чем память.

Значимые типы часто имеют совершенно разные правила, определяющие их время жизни — например, некоторые значения локальных переменных существуют только до тех пор, пока работает содержащий их метод. Тем не

¹ В этой главе аббревиатура GC используется для обозначения как *механизма сборки мусора*, так и *сборки мусора*, которую делает сборщик мусора.

менее иногда значимые типы действуют как ссылочные типы и в конечном итоге управляются сборщиком мусора. Я поясню, как это может оказаться полезным, и я расскажу о механизме *упаковки*, который делает это возможным.

Сборка мусора

CLR поддерживает работу *кучи* — сервиса, который предоставляет память для объектов и значений, временем жизни которых управляет сборщик мусора. Каждый раз, когда вы создаете экземпляр класса с помощью `new` или создаете новый объект массива, CLR выделяет новый блок кучи. Когда нужно освободить этот блок, решает сборщик мусора.

Блок кучи содержит все нестатические поля объекта или все элементы, если это массив. CLR также добавляет заголовок, который вашей программе непосредственно не виден. Он включает в себя указатель на структуру, описывающую тип объекта, и предназначен для поддержки операций, которые зависят от реального типа объекта. Например, если вы вызываете `GetType` для ссылки, среда выполнения использует этот указатель для определения типа. (Тип часто не полностью определяется статическим типом ссылки, которая может быть указателем на тип интерфейса или базовый класс фактического типа.) Он также используется для определения, какой метод использовать при вызове виртуального метода или члена интерфейса. CLR также использует заголовок, чтобы узнать, насколько велик блок кучи, — заголовок не включает размер блока, потому что среда выполнения может рассчитать его исходя из типа объекта. (Большинство типов имеют фиксированный размер. Есть только два исключения — строки и массивы, которые CLR обрабатывает как особые случаи.) Заголовок содержит еще одно поле, которое используется для ряда задач, включая многопоточную синхронизацию и генерацию хеш-кода по умолчанию. Заголовки блоков кучи — это просто деталь реализации, так что другие реализации CLI могут выбирать иные стратегии. Тем не менее будет полезно узнать об издержках. В 32-битной системе заголовок имеет длину 8 байт, а если вы работаете в 64-битном процессе, то он занимает 16. Таким образом, объект, содержащий только одно поле типа `double` (8-байтовый тип), будет использовать 16 байт в 32-битном процессе и 24 байта в 64-битном процессе.

Хотя объекты (т. е. экземпляры класса) всегда располагаются в куче, экземпляры значимых типов ведут себя по-другому: некоторые располагаются в куче, а некоторые нет. Например, CLR хранит ряд локальных переменных

значимых типов в стеке, но если значение находится в поле экземпляра класса (а экземпляр класса располагается в куче), то это значение будет расположено внутри этого объекта в куче². А в некоторых случаях значение будет занимать целый блок кучи.

Если вы обращаетесь к чему-то через переменную ссылочного типа, то вы обращаетесь к чему-то в куче. Важно уточнить, что именно я имею в виду под переменной ссылочного типа, потому что, к сожалению, терминология здесь немного запутанная: C# использует термин *ссылка* для описания двух совершенно разных вещей. В контексте этого обсуждения ссылка — это то, что можно сохранить в переменной типа, который является производным от `object`, но не от `ValueType`. В это число не входит каждый аргумент метода вида `in`-, `out`- или `ref`-, переменная ссылочного типа или возвращаемое значение. Хотя все это своего рода ссылки, аргумент `ref int` является ссылкой на значимый тип, а это не то же самое, что ссылочный тип. (CLR фактически использует другой термин для механизма, который поддерживает `ref`, `in` и `out`: он называет это *управляемыми указателями*, тем самым давая понять, что они довольно сильно отличаются от ссылок на объекты.)

Модель управляемого выполнения, используемая C# (и всеми языками .NET), означает, что CLR знает о каждом блоке кучи, который создает ваш код, а также о каждом поле, переменной и элементе массива, в которых ваша программа хранит ссылки. Эта информация позволяет среде выполнения в любой момент определить, какие объекты *достижимы*, т. е. к которым программа может получить доступ, чтобы использовать их поля и другие элементы. Если объект недостижим, то программа по определению никогда не сможет использовать его снова. Чтобы проиллюстрировать, как CLR определяет достижимость, я написал простой метод, показанный в листинге 7.1, который выбирает веб-страницы с веб-сайта моего работодателя.

Листинг 7.1. Использование и выброс объектов

```
public static string WriteUrl(string relativeUri)
{
    var baseUri = new Uri("https://endjin.com/");
    var fullUri = new Uri(baseUri, relativeUri);
    var w = new WebClient();
    return w.DownloadString(fullUri);
}
```

² Значимые типы, определенные с помощью `ref struct`, являются исключением: они всегда находятся в стеке (см. главу 18).



Обычно тип `WebClient`, показанный в листинге 7.1, не используют. В большинстве случаев вы бы задействовали более свежий `HttpClient`. Я сознательно не использую здесь `HttpClient`, потому что он содержит только асинхронные методы, что делает время жизни переменных не таким очевидным.

CLR анализирует способ использования локальных переменных и аргументов метода. Например, хотя аргумент `relativeUri` находится в области доступности всего метода, мы используем его лишь один раз в качестве аргумента при создании второго `Uri`. Переменная считается *действительной* (*live*) от точки, где она получает значение, до точки, в которой она в последний раз используется. Аргументы метода являются действительными от начала метода и до последнего использования, кроме тех случаев, когда они не используются вообще и, следовательно, никогда не являются действительными. Локальные переменные становятся действительными позже; `baseUri` становится действительной после того, как ей было присвоено начальное значение, а затем перестает быть таковой с последним использованием, что в этом примере происходит в той же точке, что и `relativeUri`. Действительность является важным свойством при определении того, используется ли конкретный объект.

Чтобы оценить роль, которую играет действительность (*liveness*), предположим, что, когда листинг 7.1 достигает строки, которая создает `WebClient`, у CLR недостаточно свободной памяти для хранения нового объекта. В этот момент он может запросить больше памяти у ОС, но у него также есть возможность попытаться освободить память от объектов, которые больше не используются, а это означает, что нашей программе не потребуется больше памяти, чем она уже использует. В следующем разделе описывается процесс, который использует CLR, когда обрабатывает второй вариант³.

Определение достижимости

CLR начинает с определения всех корневых ссылок в вашей программе. Корень — это место хранения, такое как локальная переменная, содержащее ссылку, которая, как известно CLR, была инициализирована. Ваша программа может использовать ее в будущем без посредничества какой-либо

³ CLR не всегда ждет, пока закончится память. Подробности чуть позже. Пока важным является то, что время от времени он будет пытаться освободить место.

другой ссылки на объект. Не все места хранения считаются корневыми. Если объект содержит поле экземпляра какого-то ссылочного типа, то это поле не является корневым, потому что перед тем, как вы сможете его использовать, вам нужно получить ссылку на содержащий объект, который, возможно, недостижим. Однако статическое поле ссылочного типа является корневой ссылкой, потому что программа может прочитать значение в этом поле в любое время — единственная ситуация, в которой это поле станет недоступным в будущем, — это когда компонент, определяющий тип, выгружен, что в большинстве случаев случается лишь при выходе из программы.

Локальные переменные и аргументы метода более интересны. Иногда они являются корневыми, а иногда нет. Это зависит от того, какая именно часть метода выполняется в данный момент. Локальная переменная или аргумент могут быть корневыми, только если в данный момент поток выполнения находится внутри области, в которой эта переменная или аргумент являются действительными. Таким образом, в листинге 7.1 `baseUri` становится корневой ссылкой только после того, как ей присвоено начальное значение, и до вызова создания второго `Uri`, что является довольно узким интервалом. Переменная `fullUri` остается корневой ссылкой чуть дольше, потому что становится действительной после получения начального значения и продолжает работать во время построения `WebClient` в следующей строке; ее действительность завершается только после вызова `DownloadString`.



Когда последнее использование переменной является аргументом в вызове метода или конструктора, она перестает существовать с началом вызова этого метода. В этот момент вызываемый метод перехватывает инициативу — в начале действительны его собственные аргументы (за исключением тех, которые он не использует). Тем не менее они обычно перестают быть таковыми до возврата метода. Это означает, что в листинге 7.1 объект, на который ссылается `fullUri`, может перестать быть доступным через корневые ссылки, прежде чем произойдет возврат из `DownloadString`.

Поскольку набор динамических переменных изменяется по мере выполнения программы, набор корневых ссылок также претерпевает изменения, поэтому CLR нужна возможность сформировать моментальный снимок соответствующего состояния программы. Детали не задокументированы, но сборщик мусора может приостановить все потоки, в которых выполня-

ется управляемый код, если это необходимо для обеспечения правильного поведения.

Действительные переменные и статические поля — не единственные элементы, которые могут быть корневыми. Временные объекты, созданные в результате вычисления выражений, должны оставаться действительными столько времени, сколько необходимо для завершения вычислений. Поэтому вполне возможно существование корневых ссылок, которые не соответствуют напрямую никаким именованным объектам в вашем коде. Есть и другие типы корневых ссылок. Например, класс `GCHandle` позволяет явно создавать новые корневые ссылки, что может быть полезно в сценариях взаимодействия, где некий неуправляемый код нуждается в доступе к определенному объекту. Существуют ситуации, в которых корневые ссылки создаются неявно. Взаимодействие с COM-объектами может создавать корневые ссылки без явного использования `GCHandle` — если CLR необходимо создать оболочку COM для одного из ваших объектов .NET, эта оболочка фактически будет корневой ссылкой. Вызовы в неуправляемый код могут также включать передачу указателей на память в куче, что будет означать, что соответствующий блок кучи должен рассматриваться как достижимый на время вызова. Спецификация CLI не содержит полного списка способов появления корневых ссылок, но общий принцип заключается в том, что они будут там, где важно обеспечить достижимость используемых объектов.

Составив полный список текущих корневых ссылок для всех потоков, сборщик мусора определяет, какие объекты могут быть доступны по этим ссылкам. Он просматривает каждую ссылку по очереди, и, если она не равна `null`, сборщик знает, что объект, на который она ссылается, достижим. Среди них могут содержаться дубликаты, так как несколько корневых ссылок могут ссылаться на один и тот же объект. В связи с этим сборщик мусора отслеживает, какие объекты он уже встречал. Для каждого нового обнаруженного объекта сборщик добавляет все поля экземпляра ссылочного типа из этого объекта в список ссылок для оценки, опять же, избавляясь от дубликатов. (Это включает в себя скрытые поля, сгенерированные компилятором, такие как поля для автоматических свойств, которые я описал в главе 3.) Это означает, что если объект достижим, это относится и ко всем объектам, на которые он ссылается. Сборщик мусора повторяет этот процесс, пока у него не закончатся ссылки для проверки. Любые объекты, которые он не обозначил достижимыми, считаются недоступными, потому что сборщик просто делает то, что делает программа: использует только те объекты, которые

доступны прямо или косвенно через ее переменные, временное локальное хранилище, статические поля и другие корневые ссылки.

Вернемся к листингу 7.1. К чему все это приведет, если CLR решит запустить сборку мусора при создании `WebClient`? Переменная `fullUri` все еще действительная, поэтому `Uri`, к которой она относится, достижима, но `baseUri` больше не действительна. Мы передали копию базовой `Uri` в конструктор для второй `Uri`, и если бы она содержала копию ссылки, то не имело бы никакого значения, что `baseUri` не является действительной; если есть способ добраться до объекта от корневой ссылки, то объект достижим. Но как видим, вторая `Uri` копии не содержит, поэтому первая `Uri`, выделенная в примере, будет считаться недоступной, а CLR будет вправе восстановить занимаемую ей память.

Одним из важных результатов определения достижимости является то, что сборщик мусора не смухают циклические ссылки. Это одна из причин, по которой .NET использует сборку мусора вместо подсчета ссылок (еще один популярный подход для автоматизации управления памятью). Если у вас есть два объекта, которые ссылаются друг на друга, схема подсчета ссылок решит, что оба объекта используются, поскольку каждый из них упоминается как минимум один раз. Но объекты могут быть недоступны — если нет других ссылок на них, приложение не может их использовать. Подсчет ссылок не способен этого обнаружить, поэтому он может стать причиной утечек памяти. Но для схемы, использующей сборщик мусора CLR, тот факт, что они ссылаются друг на друга, не имеет значения — сборщик мусора никогда не доберется ни до одного из них, поэтому он правильно определит, что они больше не используются.

Случайное нарушение работы сборщика мусора

Хотя сборщик мусора может обнаружить пути, которыми программа может добраться до объекта, он не может доказать, что это обязательно произойдет. Взгляните на поразительно идиотский кусок кода в листинге 7.2. Хотя вы никогда не написали бы такой плохой код, в нем содержится распространенная ошибка. Эта проблема обычно вызывается более изощренными способами, но сначала я хочу указать на нее в более очевидном примере. Показав, как этот код предотвращает освобождение неиспользуемых объектов сборщиком мусора, я опишу не такой простой, но более реалистичный сценарий, при котором часто возникает такая же проблема.

Код суммирует числа от 1 до 100 000, а затем отображает их среднее арифметическое. Первая ошибка здесь в том, что даже не нужно делать сложение в цикле, потому что есть простое и хорошо известное решение в виде формулы: $n * (n + 1) / 2$, где n в нашем случае равно 100 000. Помимо этой математической оплошности код делает нечто еще более глупое: создает список, содержащий каждое добавляемое число, однако все, что он делает с этим списком, — это извлекает его свойство `Count` для вычисления среднего значения в конце. Но что еще хуже, перед тем, как поместить его в список, код преобразует каждое число в строку, но по сути никогда не использует эти строки.

Листинг 7.2. Крайне неэффективный кусок кода

```
static void Main(string[] args)
{
    var numbers = new List<string>();
    long total = 0;
    for (int i = 1; i < 100_000; ++i)
    {
        numbers.Add(i.ToString());
        total += i;
    }
    Console.WriteLine("Total: {0}, average: {1}",
                      total, total / numbers.Count);
}
```

Очевидно, что это надуманный пример, и хотел бы я сказать, что никогда не встречал подобную нелепицу в реальных программах. К сожалению, в жизни я сталкивался с такими примерами, и они были гораздо более запутаны — когда вы встречаете такое, то требуется около получаса, чтобы понять, правда ли этот код такой ошеломляюще бесполезный. Но я это пишу не для того, чтобы пожаловаться на стандарты разработки ПО. Цель этого примера — показать, как вы можете столкнуться с пределом возможностей сборщика мусора.

Предположим, что цикл в листинге 7.2 работает какое-то время — возможно, он находится на своей 90 000-й итерации и пытается добавить запись в список `numbers`. Предположим, что `List<string>` задействовал всю свою свободную емкость, и поэтому метод `Add` должен будет выделить новый, еще больший внутренний массив. В этот момент CLR может принять решение запустить сборку мусора, чтобы выяснить, можно ли освободить место. Что же произойдет?

Листинг 7.2 создает три вида объектов: создает `List<string>` в начале цикла, создает новую `string`, каждый проход цикла вызывая `ToString()` для `int`, и, что не так очевидно, `List<string>` определяет `string[]` для хранения ссылок на эти строки. Поскольку мы продолжаем добавлять новые элементы, ему придется выделять все большие и большие массивы. (Этот массив является деталью реализации `List<string>`, поэтому напрямую его не увидеть.) Таким образом, вопрос в том, от каких из этих объектов сборщик мусора может избавиться, чтобы освободить место для большего массива при вызове `Add?`

Наша переменная `numbers` остается действительной до последнего оператора программы, а мы смотрим на более раннюю точку кода, поэтому объект `List<string>`, на который она указывает, достижим. Массив `string[]`, который она в данное время использует, также должен быть достижимым: он выделяет более новый, более крупный объект, но ему нужно будет скопировать содержимое старого объекта в новый, поэтому список все еще должен хранить в одном из своих полей ссылку на этот текущий массив. Поскольку этот массив все еще достижим, то каждая строка, на которую ссылается массив, также будет достижима. Наша программа уже создала 90 тысяч строк, и сборщик обнаружит их все, начав с нашей переменной `numbers` и просматривая поля объекта `List<string>`, на который она ссылается, а затем просматривая каждый элемент в массиве, на который ссылается одно из `private` полей списка.

Единственные выделенные элементы, которые сборщик мусора может собрать, — это старые массивы `string[]`, которые `List<string>` создавал, когда список был меньше, и на которые он больше не имеет ссылки. К тому времени, когда мы добавим 90 тысяч элементов, список, вероятно, несколько раз изменится в размерах. Таким образом, в зависимости от того, когда сборщик мусора последний раз запускался, он, вероятно, сможет найти несколько таких неиспользуемых массивов. Но интереснее то, чего он не может освободить.

Программа никогда не использует 90 тысяч строк, которые создает, поэтому в идеале мы хотели бы, чтобы сборщик мусора освободил занимаемую ими память, так как они будут занимать несколько мегабайтов. Из-за того что программа такая короткая, легко увидеть, что эти строки не используются. Но сборщик не будет об этом знать; он основывает свои решения на достижимости и, начав с переменной `numbers`, правильно определяет, что все 90 тысяч строк достижимы. Как известно сборщику мусора, вполне

возможно, что свойство `Count` списка, которое мы используем после завершения цикла, будет обращаться к содержимому списка. Мы с вами знаем, что это не так, потому что в этом нет необходимости, но лишь потому, что мы знаем, что означает свойство `Count`. Чтобы сборщик мог сделать вывод, что программа никогда не будет использовать какие-либо элементы списка прямо или косвенно, ему необходимо знать, что `List<string>` делает внутри своих методов `Add` и `Count`. Это будет означать анализ с уровнем детализации, намного превышающим описанные мной механизмы, что может сделать сборку мусора гораздо более затратной. Более того, даже при серьезном повышении сложности, необходимом для определения того, какие достижимые объекты этот пример никогда не будет использовать, в более реалистичных сценариях сборщик вряд ли сможет делать более достоверные прогнозы, чем те, которые основаны на достижимости.

Например, гораздо более вероятно, что вы столкнетесь с этой проблемой в кэше. Если вы напишите класс, который кэширует данные, которые затратно получать или рассчитывать, то представьте себе, что произойдет, если ваш код только добавляет элементы в кэш и никогда не удаляет их. Все кэшированные данные будут доступны до тех пор, пока доступен сам объект кэша. Проблема заключается в том, что ваш кэш будет занимать все больше и больше места, и если на вашем компьютере недостаточно памяти для хранения всех фрагментов данных, которые ваша программа, возможно, захочет использовать, в конечном итоге он исчерпает память.

Наивный разработчик будет жаловаться, что это проблемы сборщика мусора — смысл сборщика мусора заключается в том, чтобы не думать об управлении памятью, так почему же у меня вдруг заканчивается память? Но, конечно, проблема в том, что у сборщика мусора нет возможности узнать, какие объекты можно безопасно удалить. Он не экстрасенс и не способен точно предсказать, какие кэшированные элементы могут понадобиться вашей программе в будущем — если код выполняется на сервере, то будущее использование кэша может зависеть от того, какие запросы получает сервер, чего сборщик мусора точно не может предсказать. Поэтому хотя можно предположить, что управление памятью действует достаточно умно, чтобы проанализировать что-то столь простое, как в листинге 7.2, в целом это не та проблема, которую способен решить сборщик мусора. Таким образом, если вы добавляете объекты в коллекции и сохраняете доступность этих коллекций, сборщик мусора будет считать все в этих коллекциях достижимым. Удаление элементов — это ваша задача.

Коллекции не единственный способ, которым вы можете обмануть сборщик мусора. Как я покажу в главе 9, существует распространенный сценарий, при котором неосторожное использование событий может вызвать утечку памяти. В более общем смысле, если ваша программа делает объект достижимым, сборщик мусора не может определить, собираетесь ли вы снова использовать этот объект, поэтому он должен быть сохранен.

Тем не менее с небольшой помощью сборщика мусора эту проблему можно сделать менее острой.

Слабые ссылки

Хотя сборщик мусора будет проходить по обычным ссылкам в полях достижимого объекта, можно сохранить слабую ссылку. Сборщик не проходит по слабым ссылкам, поэтому, если единственный способ достичь объекта — это слабые ссылки, сборщик ведет себя так, как будто объект недоступен, и удаляет его. Слабая ссылка — это способ сообщить CLR: «Не ориентируйся на меня при сохранении этого объекта, но пока он еще кому-то нужен, я бы хотел иметь к нему доступ». В листинге 7.3 показан кэш, где используется `WeakReference<T>`.

Листинг 7.3. Использование слабых ссылок в кэше

```
public class WeakCache<TKey, TValue> where TValue : class
{
    private readonly Dictionary<TKey, WeakReference<TValue>> _cache =
        new Dictionary<TKey, WeakReference<TValue>>();

    public void Add(TKey key, TValue value)
    {
        _cache.Add(key, new WeakReference<TValue>(value));
    }

    public bool TryGetValue(TKey key, out TValue cachedItem)
    {
        WeakReference<TValue> entry;
        if (_cache.TryGetValue(key, out entry))
        {
            bool isAlive = entry.TryGetTarget(out cachedItem);
            if (!isAlive)
            {
                _cache.Remove(key);
            }
        }
    }
}
```

```
        return isAlive;
    }
else
{
    cachedItem = null;
    return false;
}
}
```

Этот кэш хранит все значения с помощью `WeakReference<T>`. Его метод `Add` передает объект, для которого мы хотели бы использовать слабую ссылку в качестве аргумента конструктора для нового объекта `WeakReference<T>`. Метод `TryGetValue` пытается получить значение, ранее сохраненное с помощью `Add`. Сначала проверяется, содержит ли словарь соответствующую запись. Если содержит, то значением этой записи будет `WeakReference<T>`, которую мы создали ранее. Мой код вызывает метод `TryGetTarget` слабой ссылки, который вернет `true`, если объект все еще доступен, и `false`, если нет.



Наличие не обязательно означает достижимость. Объект мог стать недоступным со времени последней сборки мусора. Или, возможно, с момента выделения объекта вообще не было сборки мусора. `TryGetTarget` заботит не то, доступен ли объект прямо сейчас, а лишь то, обнаружил ли сборщик мусора, что он подлежит освобождению.

Если объект доступен, `TryGetTarget` предоставляет его через параметр `out`, и это будет строгой ссылкой. Таким образом, если метод возвращает значение `true`, нам не нужно беспокоиться о том, что через мгновение объект может стать недоступным, — тот факт, что мы теперь сохранили эту ссылку в переменной, которую вызывающая сторона передала через аргумент `cachedItem`, сохранит и сам объект. Если `TryGetTarget` возвращает `false`, мой код удаляет соответствующую запись из словаря, поскольку она представляет объект, которого больше не существует. Это важно, потому что даже если слабая ссылка не сохранит свою цель, `WeakReference<T>` — это отдельный объект, и сборщик не может освободить его, пока я не удалю его из этого словаря. Листинг 7.4 задействует этот код, вызывая пару сборок мусора, чтобы мы могли увидеть работу сборщика в действии. (Каждый этап разделен на отдельные методы с отключенным встраиванием, потому что в противном случае JIT-компилятор в .NET Core встроит эти методы и в итоге создаст

скрытые временные переменные, которые могут сделать массив достижимым дольше, чем следовало бы, тем самым исказив результаты теста.)

Листинг 7.4. Пробуем слабый кэш

```
class Program
{
    static WeakCache<string, byte[]> cache =
        new WeakCache<string, byte[]>();
    static byte[] data = new byte[100];

    static void Main(string[] args)
    {
        AddData();
        CheckStillAvailable();

        GC.Collect();
        CheckStillAvailable();

        SetOnlyRootToNull();
        GC.Collect();
        CheckNoLongerAvailable();
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void AddData()
    {
        cache.Add("d", data);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void CheckStillAvailable()
    {
        Console.WriteLine("Retrieval: " +
            cache.TryGetValue("d", out byte[] fromCache));
        Console.WriteLine("Same ref? " +
            object.ReferenceEquals(data, fromCache));
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void SetOnlyRootToNull()
    {
        data = null;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void CheckNoLongerAvailable()
```

```
{  
    byte[] fromCache;  
    Console.WriteLine("Retrieval: " + cache.TryGetValue("d",  
                                              out fromCache));  
    Console.WriteLine("Null? " + (fromCache == null));  
}  
}
```

Все начинается с создания экземпляра моего класса кэша и последующего добавления в кэш ссылки на 100-байтовый массив. Листинг также хранит ссылку на тот же массив в статическом поле `data`, сохраняя его достижимым, пока код не вызовет `SetOnlyRootToNull`, который устанавливает его значение равным `null`. В этом примере мы пытаемся извлечь значение из кэша сразу после его добавления, а также использовать `object.ReferenceEquals` для простой проверки того, что возвращаемое значение действительно ссылается на тот же объект, который мы туда поместили. Затем я запускаю сборку мусора и пытаюсь проделать это снова. (Такой вид искусственного тестового кода является одной из немногих ситуаций, в которых потребуется это делать, — подробности см. в разделе «Принудительная сборка мусора» на с. 410.) Поскольку поле `data` по-прежнему содержит ссылку на массив, то он все еще доступен, поэтому мы ожидаем, что значение все еще будет доступно для получения из кэша. Затем я присваиваю `data` значение `null`, поэтому мой код перестает быть ответственным за достижимость массива. Единственная оставшаяся ссылка — это слабая ссылка, поэтому, когда я запускаю другую сборку мусора, я ожидаю, что массив будет освобожден и последний поиск в кэше завершится неудачей. Чтобы убедиться в этом, я проверяю как возвращаемое значение, ожидая `false`, так и значение, возвращаемое через параметр `out`, который должен быть `null`. И это именно то, что происходит, когда я запускаю программу:

```
Retrieval: True  
Same ref? True  
Retrieval: True  
Same ref? True  
Retrieval: False  
Null? True
```

Позже я опишу финализацию, которая усложняет ситуацию и создает полограничное состояние, в котором объект уже был отмечен как недоступный, но еще не исчез. Объекты, находящиеся в таком состоянии, как правило, бесполезны, поэтому по умолчанию слабая ссылка будет считать ожидающие финализации объекты уже удаленными. Это называется короткой

слабой ссылкой. Если по какой-то причине вам нужно узнать, действительно ли объект исчез (а не всего лишь ожидает этого), то конструктор класса `WeakReference<T>` имеет перегрузки, часть из которых умеют создавать длинную слабую ссылку, обеспечивающую доступ к объекту даже в этом промежутке между недоступностью и окончательным удалением.



Написание кода, иллюстрирующего поведение сборщика мусора, — это шаг на кривую дорожку. Принципы работы остаются прежними, но точное поведение небольших примеров с течением времени меняется. (Мне пришлось изменить некоторые примеры из предыдущих изданий книги.) Вполне возможно, что если вы запустите эти примеры, то увидите другое поведение из-за изменений в среде выполнения после выхода книги в печать.

Освобождение памяти

До сих пор я описывал лишь то, как CLR определяет, какие объекты больше не используются, но не то, что происходит после этого. Выявив мусор, среда выполнения должна его убрать. Для маленьких и больших объектов CLR использует разные стратегии. (По умолчанию большой объект — это тот, размер которого превышает 85 тысяч байт.) Большинство выделений памяти обслуживают небольшие объекты, поэтому сначала я расскажу о них.

CLR пытается сохранить непрерывность свободного пространства кучи. Очевидно, что это легко, когда приложение запускается впервые, потому что нет ничего, кроме свободного места, и поддерживать непрерывность можно достаточно легко, выделяя память для каждого нового объекта непосредственно после последнего. Но после первой сборки мусора куча уже вряд ли будет выглядеть так аккуратно. Большинство объектов имеют короткий срок жизни и становятся недостижимыми между двумя последовательными сборками мусора. Однако некоторые из них все еще могут использоваться. Время от времени приложения создают объекты, которые задерживаются дольше. Какая бы работа ни происходила во время сборки мусора, в ней, вероятно, будут использоваться какие-то объекты, так что последние выделенные блоки кучи, скорее всего, все еще будут использоваться. Это означает, что конец кучи может выглядеть примерно так, как показано на рис. 7.1, где серые прямоугольники являются достижимыми блоками, а белые показывают блоки, которые больше не используются.

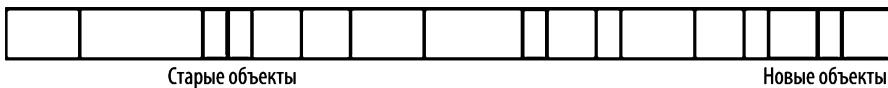


Рис. 7.1. Раздел кучи с некоторыми достижимыми объектами

Одна из возможных стратегий распределения памяти — начать использовать эти пустые блоки, когда требуется новая память, но у этого подхода есть пара недостатков. Во-первых, как правило, он достаточно расточительный, потому что блоки, которые требуются приложению, вряд ли будут точно соответствовать имеющимся просветам. Во-вторых, поиск подходящего пустого блока может быть затратным, особенно если есть много просветов, а вы пытаетесь выбрать тот, который минимизирует потери. Конечно, это не сверхзатратно — многие кучи работают таким образом, — но это намного дороже, чем исходная ситуация, когда каждый новый блок может быть размещен непосредственно после последнего, потому что все свободное место пока непрерывно. Затраты из-за фрагментации кучи вполне ощутимы, поэтому CLR обычно пытается вернуть кучу в состояние, когда свободное пространство непрерывно. Как показано на рис. 7.2, она перемещает все достижимые объекты к началу кучи, так что все свободное пространство оказывается в конце, что возвращает благоприятную ситуацию, позволяющую выделять новые блоки кучи один за другим в непрерывном разделе свободного пространства.

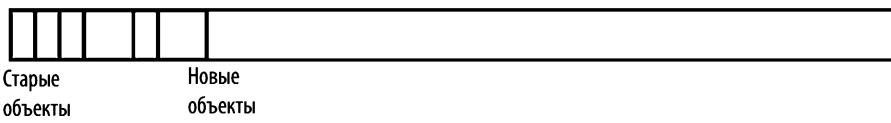


Рис. 7.2. Раздел кучи после уплотнения

Среда выполнения должна гарантировать, что ссылки на эти перемещенные блоки остаются рабочими и после перемещения блоков. CLR реализует ссылки как указатели (хотя для спецификации CLI этого не требуется, так как ссылка — это просто значение, которое указывает на какой-то конкретный экземпляр в куче). Она уже знает, где находятся все ссылки на какой-либо конкретный блок, потому что ей уже пришлось найти их, чтобы узнать, какие блоки доступны. При перемещении блока CLR корректирует все эти указатели.

Помимо того что выделение блоков кучи становится относительно дешевой операцией, сжатие обеспечивает еще одно преимущество в производительности. Поскольку блоки размещаются в смежной области свободного пространства, объекты, которые были созданы в быстрой последовательности, обычно оказываются в куче рядом друг с другом. Это важно, потому что кэши в современных процессорах, как правило, предпочитают сосредоточенность (т. е. работают лучше, когда связанные части данных хранятся близко друг к другу).

Низкая стоимость размещения и высокая вероятность хорошей сосредоточенности могут иногда означать, что кучи со сборкой мусора обладают лучшей производительностью, чем традиционные, которые требуют от программы явного освобождения памяти. Это может показаться удивительным, учитывая, что сборщик мусора на первый взгляд выполняет много дополнительной работы, которая не нужна в куче без сборки мусора. Однако некоторая часть этой «дополнительной работы» иллюзорна — ведь как-то нужно отслеживать, какие объекты еще используются, а традиционные кучи просто передают эти дополнительные расходы в наш код. Однако перемещение имеющихся блоков памяти обходится дорого, поэтому CLR использует ряд приемов, чтобы снизить объем требуемого копирования.

Чем старше объект, тем дороже будет для CLR сжимать кучу, когда он наконец станет недостижимым. Если во время работы сборщика мусора последний выделенный объект оказывается недостижим, уплотнение для него бесплатное: после него больше нет объектов, поэтому ничего не нужно перемещать. Сравните это с первым объектом, который выделяет ваша программа, — если станет недоступным он, то уплотнение будет означать перемещение каждого достижимого объекта в куче. В целом, чем старше объект, тем больше объектов будет помещено после него, поэтому при уплотнении потребуется перемещать больше данных. Копирование 20 МБ данных для сохранения 20 байт — не самый взаимовыгодный обмен. Поэтому CLR часто откладывает уплотнение старых фрагментов кучи.

Чтобы решить, что же считать «старым», CLR делит кучу на *поколения*. Границы между поколениями смещаются при каждой сборке мусора, потому что поколения определяются тем, сколько сборок мусора пережил объект. Любой объект, выделенный после самой последней сборки мусора, относится к поколению 0, поскольку он еще не пережил ни одной сборки. При следующем запуске сборки мусора достижимые объекты поколения 0 будут перемещены ввиду необходимости уплотнения кучи, после чего будут считаться поколением 1.

Объекты поколения 1 еще не считаются старыми. Сборка мусора, как правило, происходит, когда код находится в процессе выполнения — в конце концов она срабатывает, когда пространство в куче израсходовано, а этого не произойдет, если программа простоявает. Таким образом, существует высокая вероятность того, что некоторые из недавно выделенных объектов участвуют в незавершенной работе, и, хотя в настоящее время они достижимы, они скоро станут недостижимыми. Поколение 1 служит своего рода зоной ожидания, где мы определяем, какие объекты окажутся недолговечными, а какие останутся надолго.

Поскольку программа продолжает выполняться, то время от времени будет запускаться и сборка мусора, повышая поколение выживших объектов до 1. Некоторые объекты поколения 1 станут недостижимыми. Однако сборщик мусора не обязательно сразу уплотняет эту часть кучи — до уплотнения поколения 1 он может проводить несколько сборок и уплотнений поколения 0, но рано или поздно это все равно произойдет. Объекты, которые выживают на этой стадии, перемещаются в поколение 2, которое служит самым старым поколением.

CLR пытается высвободить память из поколения 2 гораздо реже, чем из остальных. Исследования показывают, что в большинстве приложений объекты, которые доживаются до поколения 2, вероятно, останутся достижимыми еще в течение длительного времени, поэтому когда один из этих объектов в конечном итоге окажется недостижимым, он, вероятно, будет очень старым, как и объекты вокруг него. Это означает, что уплотнение этой части кучи для освобождения памяти является дорогостоящим по двум причинам: скорее всего, за этим старым объектом не только следует большое количество других объектов (требующих копирования большого объема данных), но и занимаемая им память, возможно, не использовалась долгое время, т. е. с большой долей вероятности больше не содержится в кэше процессора, что еще больше замедляет копирование. Затраты на кэширование будут иметь эффект и после сборки мусора, потому что если ЦП пришлось переносить мегабайты данных вокруг старых областей кучи, это, вероятно, вызовет побочные эффекты при удалении других данных из кэша ЦП.

Размеры кэша могут составлять всего 512 Кбайтов при низком энергопотреблении и низкой стоимости оборудования, а могут превышать 30 Мбайт в высокопроизводительных серверно ориентированных чипах, но средний диапазон — от 2 МБ до 16 Кбайт кэша — является довольно типичным, так что куча многих приложений .NET будет больше минимального размера.

Большая часть используемых приложением данных находилась в кэше вплоть до сборки мусора в поколении 2, но исчезла бы после ее завершения. Поэтому, когда сборщик мусора завершает работу и возобновляется нормальное выполнение, код некоторое время будет работать в замедленном режиме, пока необходимые приложению данные не вернутся в кэш.

Поколения 0 и 1 иногда называют *эфемерными* поколениями, потому что они в основном и содержат объекты, которые существуют только в течение недолгого времени. Поскольку доступ к ним осуществлялся совсем недавно, содержимое этих частей кучи часто оказывается в кэше ЦП, поэтому уплотнение этих разделов не слишком затратно. Более того, поскольку у большинства объектов короткое время жизни, значительная часть памяти, которую способен освободить сборщик мусора, будет принадлежать объектам этих первых двух поколений. В связи с этим они, вероятно, будут приносить больше выгоды (в плане освобожденной памяти) в обмен на затраченное процессорное время. Таким образом, в работающей программе часто можно наблюдать несколько сборок мусора в эфемерных поколениях в секунду, тогда как между последовательными сборками поколения 2 может пройти несколько минут.

У CLR есть еще одна хитрость при работе с объектами поколения 2. Часто они не сильно изменяются, поэтому существует высокая вероятность того, что на первом этапе сборки мусора, когда среда выполнения ищет достижимые объекты, часть выполнявшейся ранее работы будет производиться повторно, поскольку CLR будет проходить по тем же ссылкам и получать те же результаты для значительных подразделов кучи. Поэтому CLR иногда использует службы защиты памяти ОС для отслеживания того, когда старые блоки кучи модифицируются. Это позволяет полагаться на обобщенные результаты предыдущих операций сборки мусора вместо того, чтобы каждый раз повторять всю работу.

Каким образом сборщик мусора решает освобождать память только поколения 0, а не поколения 1 или даже 2? Сборка мусора для всех трех поколений запускается при использовании определенного объема памяти. Таким образом, при распределении памяти для поколения 0, после того как вы выделили определенное количество байтов со времени последней сборки мусора, стартует новый цикл сборки мусора. Объекты, которые его переживут, перейдут в поколение 1, а CLR отслеживает количество байтов, добавленных в поколение 1 с момента последнего сборки 1-го поколения; если это число превышает пороговое значение, поколение 1 также будет собрано.

Поколение 2 работает так же. Пороговые значения не задокументированы и не постоянны; CLR отслеживает ваши схемы распределения памяти и изменяет эти пороговые значения в попытках отыскать баланс в эффективном использовании памяти, минимизации времени, затрачиваемого процессором на сборку мусора, и избегании чрезмерной задержки, которая может возникнуть, если CLR ждет слишком долго, что приводит к огромному объему работы, когда сборка наконец происходит.



Это объясняет упомянутый ранее факт, что CLR не обязательно ждет фактической нехватки памяти, прежде чем запускать сборку мусора. Может оказаться полезным запустить его раньше.

Вы можете задаться вопросом, а что из всей этой информации имеет практическое значение? В конце концов, суть в том, что CLR гарантирует, что блоки кучи хранятся до тех пор, пока они достижимы, и что через некоторое время после того, как они станут недоступными, она в конечном итоге освободит память и наиболее эффективно использует разработанную для этого стратегию. Нужны ли детали этой схемы оптимизации поколений разработчику? До тех пор пока одни методы кодирования считаются более эффективными, чем другие, — да, нужны.

Самым очевидным выводом является то, что чем больше объектов вы выделите, тем сложнее будет работать сборщику мусора. Но об этом вы, вероятно, догадались бы и без деталей реализации. А вот что менее заметно, так это то, что большие объекты означают больше работы для сборщика мусора, так как запуск сборки мусора для каждого поколения зависит от количества использованной памяти. Таким образом, большие объекты не только увеличивают нагрузку на память, но и потребляют больше процессорного времени в результате более частого запуска сборщика мусора.

Возможно, самым важным фактом, следующим из понимания работы основанного на поколениях сборщика мусора, будет то, что на объем его работы влияет время жизни объекта. Объекты, которые живут очень короткое время, обрабатываются эффективно, потому что используемая ими память быстро освобождается во время сборки мусора в поколении 0 или 1, а объем данных, которые необходимо переместить для уплотнения кучи, оказывается небольшим. С объектами, которые живут очень долго, тоже все хорошо, потому что они окажутся в поколении 2. Они не будут часто перемещаться, потому что в этой части кучи сборка мусора проходит нечасто. Кроме того,

CLR может использовать функцию обнаружения записи диспетчера памяти ОС для более эффективного обнаружения достижимости старых объектов. Хотя очень недолговечные и очень долгоживущие объекты обрабатываются эффективно, объекты, которые живут достаточно долго, чтобы попасть в поколение 2, но не задерживаются там надолго, являются проблемой. Microsoft иногда называет это *кризисом среднего возраста* (*mid-life crisis*).

Если в вашем приложении много объектов, доживающих до 2-го поколения, но затем становящихся недостижимыми, CLR потребуется выполнять сборку мусора во 2-м поколении чаще, чем это было бы в других случаях. (Фактически поколение 2 собирается только во время полной сборки, которая также освобождает пространство, ранее использовавшееся большими объектами.) Как правило, она значительно затратнее, чем сборка в других поколениях. При работе со старыми объектами уплотнение обходится дороже, а кроме того, требуется больше уборки при нарушении целостности кучи второго поколения. Карту достижимости, которую CLR создал для этого раздела кучи, возможно, потребуется перестроить, а сборщику мусора на время уплотнения необходимо будет отключить используемое для этого обнаружение записи, что повлечет за собой затраты. Есть большая вероятность, что львиная доля этой части кучи также окажется вне кэша процессора, поэтому работа с ней может быть медленной.

Полная сборка мусора потребляет значительно больше процессорного времени, чем сборки в эфемерных поколениях. В приложениях с пользовательским интерфейсом это может вызывать задержки достаточно длительные, чтобы раздражать пользователя, особенно если части кучи были выгружены ОС. В серверных приложениях полные сборки могут вызывать значительные скачки относительно среднего времени, необходимого для обслуживания запроса. Такие проблемы — это, само собой, не конец света, и, как я опишу позже, CLR содержит кое-какие механизмы для сглаживания подобных моментов. Тем не менее минимизация количества объектов, которые доживают до поколения 2, благотворно сказывается на производительности. Вы должны учитывать это при разработке кода, который кэширует данные в памяти, — политика устаревания кэша, которая не учитывает сборку мусора, может работать неэффективно, и если вы не знаете об опасности, которую представляют собой объекты среднего возраста, бывает трудно понять почему. Кроме того, как я покажу позже в этой главе, проблема кризиса среднего возраста — это одна из причин, по которой вы можете захотеть по возможности избегать деструкторов C#.

Кстати, я опустил некоторые подробности работы кучи. Например, я не сказал ни о том, как сборщик мусора выделяет разделы адресного пространства для кучи в кусках фиксированного размера, ни о том, как он фиксирует и освобождает память. Хотя все эти нюансы и интересны, они имеют гораздо меньшее отношение к тому, как вы разрабатываете свой код, нежели понимание того, какие предположения делает сборщик мусора относительно времени жизни обычных объектов.

Говоря об освобождении памяти, занятой недостижимыми объектами, следует упомянуть еще одну вещь. Как упоминалось ранее, крупные объекты работают по-разному. Существует отдельная куча под названием *куча больших объектов* (Large Object Heap, LOH), и CLR использует ее для любого объекта размером более 85 тысяч байт. Именно сам объект, а не общий объем памяти, выделяемой объектом во время конструирования⁴. Экземпляр класса `GreedyObject` в листинге 7.5 был бы крошечным — ему нужно лишь место для одной ссылки плюс затраты на сам блок кучи. В 32-битном процессе это будут 4 байта на ссылку и 8 байт служебной информации, а в 64-битном процессе — в два раза больше. Однако массив, на который он ссылается, имеет длину 400 000 байт, так что он отправится в кучу больших объектов, тогда как сам `GreedyObject` будет работать в обычной.

Листинг 7.5. Маленький объект с большим массивом

```
public class GreedyObject
{
    public int[] MyData = new int[100000];
}
```

Технически возможно создать класс, экземпляры которого достаточно велики, чтобы попасть в кучу больших объектов, но, скорее всего, это будет генерированный код или специально созданный пример. На практике большинство блоков кучи больших объектов будут содержать массивы и, возможно, строки.

Самое большое различие между ней и обычной кучей состоит в том, что сборщик мусора обычно не уплотняет кучу больших объектов, потому что копирование больших объектов затратно. (Приложения могут потребовать уплотнения кучи объектов при следующей полной сборке мусора. Но в текущих реализациях CLR куча больших объектов приложений, которые этого

⁴ NET Core предоставляет параметр конфигурации, который позволяет вам изменить этот порог.

явно не запрашивают, никогда не уплотняется.) Она работает больше как традиционная куча C: CLR хранит список свободных блоков и, основываясь на запрошенном размере, решает, какой блок использовать. Однако список свободных блоков заполняется тем же механизмом недостижимости, который используется остальной частью кучи.

Режимы сборщика мусора

Хотя CLR настраивает некоторые аспекты поведения сборщика мусора во время выполнения (например, путем динамической регулировки пороговых значений, которые запускают сборки для каждого поколения), она также предлагает настраиваемый выбор между несколькими режимами, которые подходят различным типам приложений. Они подразделяются на две большие категории — для рабочей станции и для сервера, и затем в каждой из них вы можете использовать фоновые или непараллельные сборки. Фоновая сборка включена по умолчанию, но режим верхнего уровня по умолчанию зависит от типа проекта: для консольных приложений и приложений, использующих библиотеку графического интерфейса, например WPF, сборщик мусора работает в режиме рабочей станции, но веб-приложения ASP.NET Core меняют его на режим сервера, как и старые веб-приложения ASP.NET. Если вы не хотите использовать настройки по умолчанию, способ настройки режима сервера зависит от того, какую форму .NET вы используете. Для .NET Core вы можете явно управлять режимом сборщика, определив свойство в вашем файле `.csproj`, как показано в листинге 7.6. Его можно поместить куда угодно внутри корневого элемента `Project`.

Листинг 7.6. Включение серверной сборки мусора в файле проекта приложения .NET Core

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

Для изменения настроек сборщика мусора в приложениях, работающих в .NET Framework, используется файл конфигурации приложения. Это XML-файлы, и они по сути являются предшественниками файлов `runtimeconfig.json`. (.NET Core перешел на JSON, чтобы избежать необходимости загружать анализатор XML при запуске приложения.) Веб-приложения, созданные для .NET Framework (т. е. не для .NET Core), обычно содержат такой файл с именем `web.config`. Тем не менее в веб-приложениях, написанных с ASP.NET, по умолчанию будет работать сборка мусора сервера, так

что добавлять этот конкретный параметр в файл `web.config` не нужно. За пределами платформы для разработки интернет-приложений ASP.NET файл конфигурации обычно называется `App.config`, и многие шаблоны проектов Visual Studio добавляют этот файл автоматически. В листинге 7.7 показан файл конфигурации, который включает режим сборки мусора сервера для приложения, работающего в .NET Framework. Соответствующие строки выделены жирным шрифтом.



Свойства файла проекта недоступны приложениям во время выполнения, поэтому `ServerGarbageCollection` заставляет систему сборки добавить параметр в файл `YourApplication.runtimeconfig.json`, который генерируется для вашего приложения. Он содержит конфигурационный раздел `Properties`, содержащий один или несколько параметров *базовой конфигурации CLR*. Включение сборки мусора сервера в файле проекта устанавливает для параметра `System.GC.Server` в этом файле конфигурации значение `true`. Все настройки сборщика мусора также задаются с помощью параметров конфигурации. С их помощью можно управлять и рядом других функций CLR, например режимом JIT-компилятора.

Листинг 7.7. Устаревшая конфигурация сборки мусора сервера (для .NET Framework)

```
<?xml version="1.0" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.8" />
  </startup>
  <runtime>
    <gcServer enabled="true" />
  </runtime>
</configuration>
```

Режимы рабочей станции разработаны для задач, с которыми обычно сталкивается клиентский код, когда процесс обычно работает либо над одной задачей, либо над небольшим количеством задач одновременно. Режим рабочей станции предлагает две вариации: неконкурентную и фоновую.

В фоновом режиме (по умолчанию) сборщик мусора минимизирует время, на которое он приостанавливает потоки во время сборки мусора. Существуют определенные фазы сборки мусора, во время которых CLR вынужден приостановить выполнение для обеспечения согласованности. При сборке из эфемерных поколений потоки будут приостановлены для большинства

операций. Обычно это безболезненно, потому что эти сборки, как правило, работают очень быстро — они занимают столько же времени, сколько ошибка страницы памяти, которая не приводит к работе с диском. (Такие неблокирующие ошибки страницы памяти происходят довольно часто и настолько быстро, что многие разработчики, похоже, даже не подозревают, что они вообще случаются.) Проблема в полных сборках, и именно их фоновый режим обрабатывает по-другому. Не вся работа, выполняемая при сборке, действительно требует приостановки всего, и фоновый режим использует это, позволяя полной (поколение 2) сборке продолжать работу в фоновом потоке, не блокируя другие потоки до окончания сборки. Это может позволить машинам с несколькими процессорными ядрами (большинство компьютеров в настоящее время) выполнять полные сборки мусора на одном ядре, в то время как другие ядра продолжают полезную работу. Это особенно полезно в приложениях с пользовательским интерфейсом, поскольку снижает вероятность того, что приложение перестает отвечать на запросы из-за сборок мусора.

Неконкурентный режим предназначен для оптимизации пропускной способности на одном процессоре с одним ядром. Он может оказаться более эффективным, потому что в обмен на меньшие задержки фоновая сборка мусора использует немного больше памяти и больше циклов ЦП для любой конкретной задачи, чем непараллельная сборка мусора. В некоторых задачах ваш код может работать быстрее, если отключить ее, установив для свойства `ConcurrentGC` значение `false` в файле проекта (или, если вы работаете в .NET Framework, добавьте `<gcConcurrent enabled="false" />` в элемент `<runtime>` своего файла конфигурации). Для большей части кода на стороне клиента наиболее важно избежать задержек, которые могут быть достаточно длинными, чтобы стать заметными пользователям. Пользователи более чувствительны к случаям, когда приложение не отвечает, нежели к неоптимальной средней загрузке ЦП. Поэтому для интерактивных приложений использование немного большего количества памяти и циклов ЦП в обмен на улучшение видимой производительности обычно является хорошим компромиссом.

Режим сервера существенно отличается от режима рабочей станции. Он доступен только при наличии нескольких аппаратных потоков, например в случае многоядерного процессора или нескольких физических процессоров. (Если вы включили сборку мусора сервера, но ваш код в конечном итоге выполняется на одноядерном компьютере, он вернется к использо-

ванию сборки мусора рабочей станции.) Кстати, его доступность не имеет никакого отношения к тому, с какой ОС вы работаете. Например, режим сервера доступен как в несерверных, так и в серверных версиях Windows, если у вас есть подходящее аппаратное обеспечение, а режим рабочей доступен станции всегда⁵. В режиме сервера каждый процессор получает свой собственный раздел кучи, поэтому, когда поток работает над собственной задачей независимо от остальной части процесса, он может выделять блоки кучи с минимальной нагрузкой. В режиме сервера CLR создает несколько потоков, предназначенных для задач сборки мусора, по одному для каждого логического процессора в машине. Они работают с более высоким приоритетом, чем обычные потоки, поэтому, когда происходит сборка мусора, все доступные ядра ЦП работают с собственными кучами, что обеспечивает лучшую пропускную способность при больших кучах, чем режим рабочей станции.



Объекты, созданные одним потоком, все еще могут быть доступны другим — логически куча все еще является единой службой. Режим сервера — это просто стратегия реализации, оптимизированная для случаев, когда все потоки работают над собственными задачами, в основном изолированно. Имейте в виду, что лучше всего это работает, если все задачи имеют одинаковые схемы распределения кучи.

С режимом сервера могут возникнуть кое-какие проблемы. Этот режим работает лучше всего, когда его использует только один процесс, поскольку он настроен на одновременное использование во время сборок всех ядер ЦП. Он также имеет тенденцию потреблять значительно больше памяти, чем режим рабочей станции. Если на одном сервере запущено несколько процессов .NET, которые это делают, борьба за ресурсы может снизить производительность. Другая проблема со сборкой мусора сервера заключается в том, что она отдает предпочтение производительности, а не времени ответа. В частности, сборки происходят реже, потому что это, как правило, увеличивает выигрыш в производительности, который дают многопроцессорные сборки, но это также означает, что каждая отдельная сборка занимает больше времени.

⁵ Хотя в наши дни одноядерные процессоры встречаются редко, все еще распространена работа на виртуальных машинах, которые представляют собой только одно ядро для исполняемого кода. Это часто происходит, если ваше приложение работает, например, в облачной службе, использующей тариф на основе потребляемого объема.

Как и в случае со сборкой рабочей станции, сборка мусора сервера по умолчанию использует фоновый режим. В некоторых случаях вы можете обнаружить, что, отключив ее, вы улучшаете производительность, однако следует опасаться проблем, которые это может вызвать. Например, продолжительность полной сборки в режиме непараллельной сборки сервера может, при большой куче, привести к серьезным задержкам в обратной связи веб-сайта. Вы можете сгладить проблему несколькими способами. Можно запрашивать уведомления незадолго до того, как произойдет сборка (используя методы класса `System.GC`, такие как `RegisterFor`, `FullGCNotification`, `WaitForFullGCApproach` и `WaitForFullGCComplete`), и если у вас есть серверный парк, то сервер, на котором запущена полная сборка мусора, может попросить балансировщик нагрузки не передавать запросы до ее завершения. Более простой вариант — оставить включенной фоновую сборку. Поскольку фоновые сборки позволяют потокам приложения продолжать работу и даже выполнять сборки 0-го и 1-го поколения, в то время как полная сборка выполняется в фоновом режиме, значительно улучшается время отклика приложения во время сборок и сохраняется выигрыш в производительности режима сервера.

Временная приостановка сборок мусора

Можно попросить .NET запретить сборку мусора во время выполнения определенного раздела кода. Это полезно, если вы выполняете задачу, чувствительную ко времени. Windows, macOS и Linux не являются ОС реального времени, поэтому никаких гарантий быть не может, но временное выключение сборщиков мусора в критические моменты может тем не менее оказаться полезным для снижения вероятности того, что все замедлится в самый нужный момент. Имейте в виду, что этот механизм работает, передвигая вперед любую работу сборщика мусора, которая в противном случае произошла бы в соответствующем разделе кода, поэтому это может привести к более ранним задержкам, связанным со сборкой мусора. Это лишь гарантирует, что после того, как обозначенная область кода начнет работать, и если выполняются определенные требования, больше не будет происходить сборок мусора. По сути, задержки устраняются прежде, чем начинается чувствительная ко времени работа.

Класс `GC` содержит метод `TryStartNoGCRegion`, который вы вызываете, чтобы указать, что вы приступаете к работе, которая не должна прерываться сборками мусора. Вы должны передать значение, указывающее, сколько

памяти вам понадобится во время этой работы, и, прежде чем продолжить, он попытается убедиться, что по крайней мере столько памяти доступно (и выполнить сборку мусора для освобождения нужного объема, если необходимо). Если метод завершается успешно, то, пока вы не потребляете больше памяти, чем запрошено, ваш код не будет прерван сборкой мусора. По окончании чувствительной ко времени задачи, следует вызвать `EndNoGCRegion`, чтобы сборщик мог вернуться к своей нормальной работе. Если перед вызовом `EndNoGCRegion` ваш код использует больше памяти, чем вы запрашивали, CLR, возможно, придется выполнить сборку мусора, но это произойдет только в том случае, если нет возможности избежать этого до вызова `EndNoGCRegion`.

Несмотря на то что форма `TryStartNoGCRegion` с одним аргументом по необходимости выполнит полную сборку мусора, некоторые перегрузки принимают `bool`, позволяя вам указать, что, если для освобождения необходимого пространства потребуется полная блокирующая сборка мусора, вы предпочтете прервать выполнение метода. Существуют также перегрузки, в которых вы можете отдельно указать свои требования к памяти в обычной куче и куче больших объектов.

Случайный срыв уплотнения

Уплотнение кучи (heap compaction) — важная функция сборщика мусора CLR, поскольку она оказывает сильное влияние на производительность. Некоторые операции могут предотвратить уплотнение, и это то, чего мы хотели бы избежать, потому что фрагментация может увеличить использование памяти и значительно снизить производительность.

Чтобы иметь возможность уплотнять кучу, CLR нужна возможность перемещать блоки кучи. Обычно это можно сделать, потому что известны все места, в которых ваше приложение ссылается на блоки кучи и есть возможность корректировать ссылки при перемещении блоков. Но что, если вы вызываете API ОС, который работает непосредственно с предоставленной вами памятью? Например, если вы читаете данные из файла или сетевого сокета, как должно происходить взаимодействие со сборкой мусора?

Если вы используете системные вызовы для чтения или записи данных с использованием таких устройств, как жесткий диск или сетевой интерфейс, они обычно работают непосредственно с памятью вашего приложения. Если вы читаете данные с диска, ОС может дать указание контроллеру диска

поместить байты непосредственно в память, которую ваше приложение передало в API. ОС выполнит необходимые вычисления для перевода виртуального адреса в физический адрес. (В случае виртуальной памяти значение, которое ваше приложение помещает в указатель, только косвенно связано с фактическим адресом в оперативной памяти вашего компьютера.) ОС заблокирует страницы на время запроса ввода-вывода, чтобы гарантировать, что физический адрес остается в силе. Затем он предоставит этот адрес дисковой системе. Это позволяет контроллеру диска копировать данные с диска непосредственно в память, не требуя дополнительного участия ЦП. Это очень эффективно, но вызывает проблемы, когда происходит уплотнение кучи. Что, если блок памяти является массивом `byte[]` в куче? Предположим, что сборка мусора происходит между просьбой прочитать данные и моментом, когда диск может их предоставить. (Шансы довольно высоки; механическому диску с вращающимися пластинами может потребоваться 10 мс или более, чтобы начать передачу данных, что с точки зрения ЦП является вечностью.) Если сборщик мусора решит переместить наш массив `byte[]` с целью уплотнения кучи, то адрес в физической памяти, который ОС предоставила контроллеру диска, окажется устаревшим, поэтому, когда контроллер начнет помещать данные в память, он будет записывать их не туда.

Есть три способа, с помощью которых CLR постараётся этого избежать. Один из них — заставить сборщик мусора подождать. Перемещение кучи может быть приостановлено на время выполнения операций ввода-вывода. Но это путь в никуда; нагруженный сервер может работать в течение нескольких дней, не переходя в состояние, в котором не выполняются операции ввода-вывода. На самом деле серверу даже не нужно быть нагруженным. Он может выделить несколько массивов `byte[]` для хранения следующих нескольких входящих сетевых запросов и, как правило, будет пытаться избежать перехода в состояние, когда у него не будет хотя бы одного такого доступного буфера. Операционная система будет иметь указатели на все это и вполне может предоставить сетевой карте соответствующий физический адрес, чтобы она могла приступить к работе, как только данные начнут поступать. Таким образом, даже на простаивающем сервере есть определенные буфера, которые нельзя перемещать.

Альтернативой для CLR является предоставление для операций такого рода отдельной неподвижной кучи. Возможно, мы могли бы выделить фиксированный блок памяти для операции ввода-вывода, а затем скопировать

результаты в массив `byte[]` в куче сборщика мусора после завершения ввода-вывода. Но это тоже не лучшее решение. Копирование данных затратно — чем больше копий входящих или исходящих данных вы делаете, тем медленнее будет работать ваш сервер. Поэтому вам и нужно, чтобы сетевое и дисковое оборудование использовало для работы с данными их настояще местоположение. И если бы эта гипотетическая фиксированная куча была чем-то большим, чем деталь реализации CLR, и была доступна коду приложения для непосредственного использования в целях минимизации копирования, это стало бы отличной возможностью совершить все те ошибки в управлении памятью, с которыми сборщик мусора призван бороться.

Так что CLR использует третий подход: выборочно предотвращает перемещение блоков кучи. Сборщик мусора может свободно работать во время операций ввода-вывода, но некоторые блоки кучи могут быть закреплены. Закрепление блока устанавливает флаг, который сообщает сборщику, что блок в настоящее время не может быть перемещен. Таким образом, если сборщик мусора встретит такой блок, он просто оставит его на месте, но попытается переместить все вокруг него.

Существует три способа, которыми код C# обычно закрепляет блоки кучи. Вы можете сделать это явно, используя ключевое слово `fixed`. Это позволяет получить необработанный указатель на место хранения, такое как поле или элемент массива, и компилятор генерирует код, который гарантирует, что, пока фиксированный указатель находится в области доступности, блок кучи, на который он ссылается, будет закрепленным. Более распространенным способом закрепления блока является межпрограммное взаимодействие (т. е. вызовы в неуправляемый код, такой как методы в компоненте COM или API OC). Если вы сделаете вызов API, для которого требуется указатель на что-либо, CLR определит, когда он указывает на блок кучи, и автоматически закрепит этот блок. (По умолчанию CLR автоматически открепляет его после возврата из метода. Если вы вызываете асинхронный API, то можете использовать класс `GCHandle`, упомянутый ранее, для закрепления блока кучи до тех пор, пока не открепите его явно.)

Третий и наиболее распространенный способ закрепления блоков кучи также является наименее прямым: многие API библиотек классов вызывают неуправляемый код от вашего имени и в результате закрепляют массивы, которые вы передаете. Например, библиотека классов определяет класс `Stream`, который представляет поток байтов. Есть несколько реализаций этого абстрактного класса. Некоторые потоки целиком работают в памяти,

но некоторые являются обертками механизмов ввода-вывода, обеспечивающая доступ к файлам или данным, отправляемым или получаемым через сетевой сокет. Абстрактный базовый класс `Stream` определяет методы для чтения и записи данных через массивы `byte[]`, а реализации потоков на основе ввода/вывода часто на необходимое время закрепляют блоки кучи, содержащие эти массивы.

Если вы пишете приложение, которое выполняет много закреплений (например, много сетевых операций ввода-вывода), вам стоит тщательно продумать, как вы выделяете память для закрепляемых массивов. Закрепление больше всего вредит недавно выделенным объектам, поскольку они находятся в области кучи, где происходит больше всего работ по уплотнению. Закрепление недавно выделенных блоков приводит к фрагментации эфемерного раздела кучи. Память, которая обычно была бы освобождена почти мгновенно, теперь должна ждать, пока блоки не будут откреплены. К тому времени, когда сборщик сможет до них добраться, после них может быть выделено множество других блоков, а это означает, что потребуется гораздо больше работы по освобождению памяти.

Если закрепление вызывает проблемы в вашем приложении, вы будете наблюдать ряд распространенных симптомов. Процент процессорного времени, затрачиваемого на сборку мусора, будет относительно высоким — все, что превышает 10%, считается плохим. Но само по себе это не обязательно следствие закреплений. Это может оказаться результатом того, что объекты среднего возраста вызывают слишком много полных сборок мусора. В этом случае можно отслеживать количество закрепленных блоков в куче⁶, пытаясь определить, являются ли они виновниками происходящего. Если окажется, что именно избыточное закрепление вызывает неприятности, есть два способа этого избежать. Один из них заключается в том, чтобы спроектировать ваше приложение так, чтобы закреплялись только блоки, которые располагаются в куче больших объектов. Помните, что по умолчанию куча больших объектов не сжимается, поэтому закрепление не требует никаких затрат — сборщик не будет перемещать блок в любом случае. Сложной частью этого

⁶ В случае .NET Core вы можете использовать бесплатный инструмент Microsoft под названием *PerfView*. В случае .NET Framework вместо этого следует использовать *Performance Monitor*, встроенный в Windows. Они могут содержать многочисленные полезные статистические данные по сборщику мусора и других действий CLR, включая процент процессорного времени, потраченного на сборку мусора, количество закрепленных объектов и количество сборок поколений 0, 1 и 2. Кроме того, бесплатный инструмент *BenchMarkDotNet* содержит функцию диагностики памяти.

является то, что вам придется выполнять все свои операции ввода-вывода с массивами длиной не менее 85 тысяч байт. Это не обязательно проблема, потому что большинство API ввода/вывода могут работать с подразделом массива. Итак, если вы действительно хотите работать, скажем, с 4096-байтовыми блоками, вы можете создать один массив, достаточно большой, чтобы вместить как минимум 21 такой блок. Вам может потребоваться код, отслеживающий, какие слоты в массиве использовались, но если это исправит проблему с производительностью, то будет стоить усилий. Типы `Span<T>` и `MemoryPool<T>`, обсуждаемые в главе 18, могут упростить такую работу с массивами. (Они также значительно облегчают работу с памятью, которая не находится в куче сборщика мусора. Иными словами, вы можете полностью отказаться от закрепления, хотя тогда вам придется взять на себя ответственность за управление соответствующей памятью.)



Если вы решите уменьшить число закреплений, пытаясь использовать кучу больших объектов, вам следует помнить, что это деталь реализации. Будущие версии .NET могут полностью избавиться от кучи больших объектов. Таким образом, вам придется возвращаться к этому аспекту вашего дизайна с каждым новым выпуском .NET.

Другой способ минимизировать влияние закрепления — убедиться, что закрепляются в основном только объекты поколения 2. Если вы выделите пул буферов и будете повторно использовать их в течение всего времени работы приложения, это будет означать, что вы закрепляете блоки, которые сборщик мусора вряд ли захочет перемещать, что позволит уплотнять эфемерные поколения в любое время. Чем раньше вы выделите память для этих буферов, тем лучше, поскольку чем старше объект, тем меньше вероятность того, что сборщик мусора захочет его переместить.

Принудительная сборка мусора

Класс `System.GC` содержит метод `Collect`, который позволяет принудительно выполнять сборку мусора. Вы можете передать число, указывающее поколение, которое вы хотите собрать, или использовать перегрузку, которая не принимает аргументов, для выполнения полной сборки. Причины вызывать `GC.Collect` возникают достаточно редко. Я упоминаю здесь об этом методе, потому что он часто упоминается в сети, что может создать видимость, что он более полезен, чем есть на самом деле.

Принудительная сборка мусора может вызывать проблемы. Сборщик мусора контролирует собственную производительность и настраивает свое поведение в соответствии со схемами выделения памяти вашим приложением. Но для того, чтобы получить точную картину того, насколько хорошо работают его текущие настройки, ему необходимо достаточное времени между сборками. Если вы заставите сборки происходить слишком часто, он не сможет подстраиваться и результат окажется двояким: сборщик мусора будет запускаться чаще, чем необходимо, но при запуске его поведение будет неоптимальным. Обе проблемы могут увеличить количество процессорного времени, затрачиваемого на сборку мусора.

Так когда же стоит вызывать принудительную сборку? Если вы знаете, что ваше приложение только что закончило некую работу и собирается бездействовать, возможно, стоит подумать о принудительной сборке. Сборки мусора обычно запускаются активностью, поэтому если вы знаете, что ваше приложение собирается перейти в спящий режим, — возможно, это служба, которая только что завершила выполнение пакетного задания и больше не будет выполнять работу в течение еще нескольких часов, — то понимаете, что оно не будет выделять новые объекты и, следовательно, не будет запускать сборщик мусора автоматически. Таким образом, принудительная сборка мусора даст возможность освободить память для ОС до того, как приложение перейдет в спящий режим. Но все же, если это ваш сценарий, возможно, стоит обратить внимание на механизмы, которые позволяют вашему процессу завершиться полностью. Существуют различные способы, которыми задания или службы, работа которых требуется лишь время от времени, могут быть полностью выгружены на время бездействия. Но, если этот метод по какой-то причине вам не подходит — возможно, ваш процесс требует высоких затрат на запуск или ему необходимо продолжать работу для получения входящих сетевых запросов — принудительная полная сборка может оказаться лучшим вариантом.

Стоит знать, что сборка мусора может запуститься без какого-либо участия со стороны вашего приложения. Когда системе не хватает памяти, Windows передает сообщение всем запущенным процессам. CLR обрабатывает это сообщение и принудительно вызывает сборщик мусора. Таким образом, даже если ваше приложение не предпринимает активных действий по освобождению памяти, память может быть в конечном итоге освобождена, если это понадобится чему-то другому в системе.

Деструкторы и финализация

CLR работает не покладая рук, чтобы выяснить, когда объекты перестают использоваться. Можно заставить ее уведомлять вас об этом — вместо простого удаления недостижимых объектов CLR может сначала сообщить объекту, что его собираются удалять. CLR называет это финализацией, а C# представляет специальный синтаксис: чтобы воспользоваться финализацией, следует написать деструктор.



Если вы пришли из C++, не дайте себя одурачить ни названием, ни схожим синтаксисом. Как вы дальше увидите, деструктор C# отличается от деструктора C++ в двух важных аспектах.

Листинг 7.8 показывает деструктор. Этот код компилируется в переопределение метода `Finalize`, который, как упоминалось в главе 6, является особым методом, определяемым базовым классом `object`. Финализаторы всегда должны вызывать базовую реализацию `Finalize`, которую они переопределяют. Чтобы мы не нарушали это правило, C# генерирует этот вызов, поэтому нельзя напрямую написать метод `Finalize`. Вы не можете написать код, который вызывает финализатор, — финализатор вызывается CLR, поэтому мы не указываем уровень доступа для деструктора.

Листинг 7.8. Класс с деструктором

```
public class LetMeKnowMineEnd
{
    ~LetMeKnowMineEnd()
    {
        Console.WriteLine("Goodbye, cruel world");
    }
}
```

CLR не гарантирует запуск финализаторов по какому-либо расписанию. Прежде всего он должен обнаружить, что объект стал недостижимым, чего не произойдет, пока не запустится сборщик мусора. Если ваша программа простоявает, этого может не происходить в течение длительного времени; сборщик запускается только тогда, когда ваша программа что-то делает или когда общесистемная нагрузка на память заставляет сборщик мусора оживать. Вполне возможно, что минуты, часы или даже дни могут пройти

между тем, как ваш объект станет недостижимым, а CLR заметит, что он стал таковым.

Даже когда CLR обнаруживает недостижимость, это не означает немедленного вызова финализатора. Финализаторы работают в выделенном потоке. Поскольку текущие версии CLR имеют только один поток финализации (независимо от того, какой режим сборщика мусора вы выбрали), медленный финализатор заставит другие финализаторы ожидать.

В большинстве случаев CLR не гарантирует запуск финализаторов вообще. Когда процесс завершается, а потоку финализации еще не удалось запустить все существующие финализаторы, он завершится, не дожидаясь их завершения.

Таким образом, финализаторы могут быть отложены на неопределенное время, если ваша программа либо простоявает, либо занята, причем без гарантий запуска вообще. Но что еще хуже — в самом финализаторе мало что можно сделать полезного.

Может показаться, что финализатор — хорошее место, чтобы убедиться, что некая работа выполнена должным образом. Например, если ваш объект записывает данные в файл, но буферизует эти данные, чтобы записать несколько больших кусков вместо крошечных фрагментов (поскольку большие фрагменты часто более эффективны), можно решить, что финализация будет хорошим выбором проверки того, что данные из ваших буферов были благополучно выгружены на диск. Но подумайте еще раз.

Во время финализации объект уже не может доверять другим объектам, на которые ссылается. Если уже работает деструктор вашего объекта, ваш объект, скорее всего, недостижим. С большой долей вероятности это означает, что любые другие объекты, на которые вы ссылаетесь, тоже стали недостижимыми. CLR часто обнаруживает недостижимость групп взаимосвязанных объектов одновременно — если ваш объект для выполнения поставленной задачи создал три или четыре объекта, весь набор станет недостижимым одновременно. CLR не дает никаких гарантий относительно порядка, в котором запускаются финализаторы. Это означает, что к моменту запуска деструктора все объекты, которые вы использовали, уже могут быть завершены. Так что, если они, в свою очередь, выполнили какую-либо завершающую очистку, их уже слишком поздно использовать. Например, класс `FileStream`, производный от `Stream` и обеспечивающий доступ к файлу, закрывает свой дескриптор файла в своем деструкторе. Таким образом, если

вы надеялись сбросить ваши данные в `FileStream`, уже слишком поздно, так как поток файла может быть уже закрыт.



Честно говоря, все немногим лучше, чем могло показаться из моего описания. Хотя CLR не гарантирует запуск большинства финализаторов, в реальности он обычно их запускает. Отсутствие гарантий имеет значение только в экстремальных ситуациях. Но это не умаляет того факта, что в целом не следует полагаться на другие объекты в своем деструкторе.

Поскольку деструкторы выглядят не очень полезными — вы понятия не имеете, будут они выполняться или нет, и не можете использовать внутри них другие объекты — тогда зачем они вообще нужны?

Основная причина существования финализации в том, что она позволяет писать типы .NET, которые являются обертками для таких вещей, которые традиционно представлены дескрипторами, например файлы и сокеты. Они создаются и управляются вне CLR — файлы и сокеты требуют выделения ресурсов ядром ОС; библиотеки также могут содержать API на основе дескрипторов, и они, как правило, выделяют память в собственных кучах для хранения информации обо всем, что представляет дескриптор. CLR не может видеть эти действия, и все, что ей доступно, — это объект .NET с полем, содержащим целое число. О том, что целое число является дескриптором ресурса вне CLR, она понятия не имеет. Поэтому ей невдомек, что важно закрывать дескриптор, когда объект выходит из использования. Здесь-то и вступают в дело финализаторы: они представляют собой место для кода, который сообщает чему-то внешнему по отношению к CLR, что сущность, представленная дескриптором, больше не используется. При таком сценарии невозможность использовать другие объекты уже не является проблемой.



Если вы пишете код, который служит оберткой для дескриптора, то следует использовать один из встроенных классов, которые являются производными от `SafeHandle`, или, если это абсолютно необходимо, создавать собственные. Этот базовый класс расширяет базовый механизм финализации некоторыми ориентированными на дескрипторы функциями, и при работе в .NET Framework он также задействует механизм, который при определенных обстоятельствах может гарантировать запуск финализатора. Кроме того, чтобы избежать преждевременного освобождения ресурсов, он получает специальную обработку уровня межпрограммного взаимодействия.

Есть несколько других применений для финализации, хотя упомянутые не-предсказуемость и ненадежность означают, что есть пределы тому, чем она может быть полезна. Некоторые классы содержат финализатор, который не делает ничего, кроме проверки того, что объект не был брошен в состоянии, в котором у него осталась незавершенная работа. Например, если вы написали класс, который буферизует данные перед записью их в файл, как описано ранее, то нужно будет определить какой-то метод, который вызывающие программы должны использовать, когда они завершат работу с вашим объектом (возможно, с именем `Flush` или `Close`). Вы можете написать финализатор, который проверяет, был ли объект переведен в безопасное состояние перед тем, как его забыть, и, если нет — выдает ошибку. Это позволяет узнать, когда программы забывают должным образом подчистить за собой. (Библиотека параллельных задач .NET, которую я опишу в главе 16, использует эту технику. Когда асинхронная операция выдает исключение, она использует финализатор, чтобы определить, когда запустившая ее программа не удосуживается обработать это исключение.)

Если вы пишете финализатор, следует отключить его, если ваш объект находится в состоянии, когда он больше не требует финализации, потому что финализация имеет свою цену. Если вы предлагаете метод `Close` или `Flush`, после их вызова финализация не требуется, поэтому следует вызвать метод `SuppressFinalize` класса `System.GC`, чтобы сборщик мусора знал, что ваш объект больше не нуждается в финализации. Если впоследствии состояние вашего объекта изменится, вы можете вызвать метод `ReRegisterForFinalize`, чтобы включить ее.



Несмотря на то что `SuppressFinalize` может избавить вас от самых серьезных затрат на финализацию, объект, который использует эту технику, все еще ведет к более высоким расходам ресурсов, чем объект без финализатора вообще. CLR выполняет дополнительную работу при создании финализируемых объектов, чтобы отслеживать те, которые еще не были завершены. (Вызов `SuppressFinalize` просто изымает ваш объект из этого списка отслеживания.) Итак, хотя подавление финализации намного лучше, чем разрешение на ее выполнение, все же лучше, если вы изначально не будете ее запрашивать.

Самые большие затраты, к которым приводит финализация, исходят из ее гарантии того, что ваш объект доживет минимум до первого поколения и, возможно, до следующего. Вспомните, все объекты, которые переживают

поколение 0, переходят в поколение 1. Если у вашего объекта есть финализатор и вы не отключили его, вызвав `SuppressFinalize`, CLR не сможет избавиться от вашего объекта, пока он не запустит свой финализатор. А поскольку финализаторы работают асинхронно в отдельном потоке, объект должен оставаться действующим, даже если он был признан недостижимым. Таким образом, объект будет невозможно собрать, даже если он уже недостижим и остается в поколении 1. Обычно вскоре после этого он оказывается завершен и, следовательно, становится пустой тратой пространства на время до сборки поколения 1. А такое случается гораздо реже, чем сборка поколения 1. Если ваш объект уже попал в поколение 1 до того, как стал недостижимым, финализатор увеличивает шансы попасть в поколение 2 непосредственно перед тем, как он вот-вот выйдет из употребления. Таким образом, завершенный объект неэффективно использует память, что является причиной, по которой следует избегать финализации, а также по возможности отключать в объектах, которые иногда ее требуют.

Немного странный итог финализации состоит в том, что недостижимый по мнению сборщика мусора объект может снова сделать себя достижимым. Можно написать деструктор, который сохраняет ссылку `this` в корневой ссылке или, допустим, в коллекции, доступной через корневую ссылку. Ничто не мешает вам сделать это, и объект продолжит работать (хотя его финализатор не будет запускаться во второй раз, если объект снова станет недостижимым), но это было бы странным поступком. Это называется воскрешением, и если вы можете это сделать, это еще не значит, что вам следует это делать. Лучше всего избегать такого.

Надеюсь, вы убедились, что деструкторы не являются обобщенным механизмом для чистого прекращения работы объектов. В основном они полезны только для работы с дескрипторами сущностей, расположенных вне контроля CLR, и лучше избегать того, чтобы полагаться на них. Если вам нужна своевременная и надежная очистка ресурсов, есть механизм получше.

IDisposable

Библиотека классов .NET определяет интерфейс с именем `IDisposable`. CLR не рассматривает его как нечто особенное, но C# имеет для него определенную встроенную поддержку. `IDisposable` – это простая абстракция; как показано в листинге 7.9, он определяет только один элемент – метод `Dispose`.

Листинг 7.9. Интерфейс IDisposable

```
public interface IDisposable
{
    void Dispose();
}
```

Идея, лежащая в основе `IDisposable`, проста. Если ваш код создает объект, который реализует этот интерфейс, вы должны вызвать `Dispose` после того, как закончили его использовать (за редким исключением — см. «Необязательное удаление» на с. 425). Это дает объекту возможность высвободить ресурсы, которые он мог выделить. Если удаляемый объект использовал ресурсы, представленные дескрипторами, он, как правило, немедленно закрывает эти дескрипторы, не дожидаясь начала финализации (и одновременно подавляет финализацию). Если объект использовал службы на некотором удаленном компьютере с сохранением состояния — возможно, удерживая соединение открытым для сервера, чтобы иметь возможность отправлять запросы, — он немедленно необходимым образом уведомляет удаленную систему о том, что больше не нуждается в ее услугах (например, закрыв соединение).



Живуч миф, что вызов `Dispose` заставляет сборщик мусора что-то делать. В интернете можно прочитать, что `Dispose` финализирует объект или даже вызывает сбор мусора. Это полная ерунда. CLR обрабатывает `IDisposable` или `Dispose` так же, как и любой другой интерфейс или метод.

`IDisposable` важен, потому что объект способен потреблять очень мало памяти, но все же связывать какие-то дорогие ресурсы. Например, рассмотрим объект, обеспечивающий соединение с базой данных. Такой объект не нуждается во многих полях — он вообще может иметь лишь одно поле, содержащее дескриптор соединения. С точки зрения CLR это довольно дешевый объект, и мы могли бы запросить сотни таких объектов без запуска сборки мусора. Но на сервере базы данных все будет выглядеть по-другому — может потребоваться значительный объем памяти для каждого входящего соединения. Соединения могут быть строго ограничены условиями лицензирования. (Это показывает, что «ресурс» — это довольно широкое понятие, и означает практически все, что может закончиться.)

Использование сборщика мусора для уведомления о том, что объекты подключения к базе данных больше не используются, и будет, скорее всего, плохой идеей. CLR будет знать, что мы выделили, скажем, 50 сущностей, но если они потребляют всего несколько сотен байтов, не будет причин для запуска сборки мусора. И все же работа нашего приложения может практически застопориться — если у нас всего 50 лицензий на подключение к базе данных, следующая попытка создать подключение окажется неудачной. И даже если таких ограничений нет, мы все можем крайне неэффективно использовать ресурсы базы данных, открывая гораздо больше соединений, чем требуется.

Крайне важно закрывать объекты подключения как можно скорее, не дожидаясь, пока сборщик мусора сообщит нам, какие из них не используются. Здесь нам и приходит на помощь `IDisposable`. Конечно, он предназначен не только для соединений с базой данных. Его использование критически важно для любого объекта, который является фасадом для чего-то, что расположено вне CLR, например файла или сетевого соединения. Даже для ресурсов, которые особо не ограничены, `IDisposable` предоставляет способ сообщить объектам, когда мы закончили работу с ними и они могут аккуратно завершить работу. Это решает описанную ранее проблему объектов, которые выполняют внутреннюю буферизацию.



Если создавать ресурс затратно, можно использовать его повторно. Это часто имеет место в случае соединений с базой данных, поэтому обычной практикой является содержание пула соединений. Вместо того чтобы закрывать соединение по окончании работы с ним, вы возвращаете его в пул, делая его доступным для повторного использования. (Большинство провайдеров доступа к данным .NET могут сделать это за вас.) Модель `IDisposable` будет полезной и в этом случае. Когда вы запрашиваете ресурсный пул для ресурса, он обычно предоставляет обертку вокруг реального ресурса, а когда вы утилизируете эту обертку, он возвращает ресурс в пул, а не освобождает его. Поэтому вызов `Dispose` — это просто способ сказать: «Я закончил с этим объектом», и реализация `IDisposable` должна решить, что делать дальше с ресурсом, который он представляет.

Реализации `IDisposable` необходимы, чтобы допускать множественные вызовы `Dispose`. Хотя это и означает, что получатели могут вызывать `Dispose` несколько раз без какого-либо вреда, не следует пытаться использовать

объект после его удаления. На самом деле библиотека классов определяет специальное исключение, которое объекты могут генерировать, если вы их подобным образом неправильно используете: `ObjectDisposedException`. (Об исключениях см. в главе 8.)

Конечно, вы можете вызывать `Dispose` напрямую, но C# также поддерживает три способа использования `IDisposable`: циклы `foreach`, операторы `using` и объявления `using`. Оператор `using` — это способ гарантировать, что вы надежно утилизируете объект, который реализует `IDisposable`, как только закончите работу с ним. Листинг 7.10 показывает, как его использовать.

Листинг 7.10. Оператор `using`

```
using (StreamReader reader = File.OpenText(@"C:\temp\File.txt"))
{
    Console.WriteLine(reader.ReadToEnd());
}
```

Этот код эквивалентен коду из листинга 7.11. Ключевые слова `try` и `finally` являются частью системы обработки исключений C#, о которой я подробно расскажу в главе 8. В данном случае они используются для обеспечения выполнения вызова `Dispose` внутри блока `finally`, даже если что-то идет не так в коде внутри блока `try`. Это также гарантирует, что вызов `Dispose` произойдет, даже если вы выполните инструкцию `return` в середине блока. (Он сработает, даже если вы используете для выхода из блока оператор `goto`.)

Листинг 7.11. Как разворачиваются операторы `using`

```
{
    StreamReader reader = File.OpenText(@"C:\temp\File.txt");
    try
    {
        Console.WriteLine(reader.ReadToEnd());
    }
    finally
    {
        if (reader != null)
        {
            ((IDisposable) reader).Dispose();
        }
    }
}
```

Если тип переменной объявления оператора `using` является значимым типом, С# не будет генерировать код, который проверяет ее на `null`, а будет просто напрямую вызывать `Dispose`. (Это так и для объявления `using`.)

В С# 8.0 добавлена более простая альтернатива — объявление `using`, показанное в листинге 7.12. Разница в том, что нам не нужно определять блок. Объявление `using` удаляет свою переменную, когда переменная выходит из области видимости. Оно по-прежнему генерирует блоки `try` и `finally`, поэтому в случаях, когда блок оператора `using` завершается в конце какого-либо другого блока (например, заканчивается в конце метода), вы можете без изменения поведения переключиться на объявление `using`. Это уменьшает количество вложенных блоков, что может облегчить чтение вашего кода. (С другой стороны, с обычным блоком `using` проще увидеть, когда объект больше не используется. Так что у каждого стиля есть свои плюсы и минусы.)

Листинг 7.12. Объявление `using`

```
using StreamReader reader = File.OpenText(@"C:\temp\File.txt");
Console.WriteLine(reader.ReadToEnd());
```

Если требуется использовать несколько удаляемых ресурсов в одной и той же области видимости, вы можете их вкладывать, но для чтения может оказаться проще, если вы составите несколько операторов `using` перед одним блоком. В листинге 7.13 таким образом реализуется копирование содержимого одного файла в другой.

Листинг 7.13. Составление операторов `using`

```
using (Stream source = File.OpenRead(@"C:\temp\File.txt"))
using (Stream copy = File.Create(@"C:\temp\Copy.txt"))
{
    source.CopyTo(copy);
}
```

Составление операторов `using` не является специальным синтаксисом; это всего лишь результат того, что за оператором `using` всегда следует один встроенный оператор, который будет выполнен до вызова метода `Dispose`. Обычно этот оператор является блоком, но в листинге 7.13 первый встроенный оператор оператора `using` является вторым оператором `using`. Если вместо этого вы используете объявления `using`, в составлении нет необходимости, потому они не требуют встроенного оператора.

Цикл `foreach` генерирует код, который будет использовать `IDisposable`, если его реализует перечислитель. В листинге 7.14 показан цикл `foreach`, который использует именно такой перечислитель.

Листинг 7.14. Цикл `foreach`

```
foreach (string file in Directory.EnumerateFiles(@"C:\temp"))
{
    Console.WriteLine(file);
}
```

Метод `EnumerateFiles` класса `Directory` возвращает `IEnumerable<string>`. Как вы видели в главе 5, в нем есть метод `GetEnumerator`, который возвращает интерфейс `IEnumerator<string>`, наследуемый от `IDisposable`. Следовательно, компилятор C# создаст код, эквивалентный листингу 7.15.

Листинг 7.15. Как разворачиваются циклы `foreach`

```
{
    I IEnumerator<string> e =
        Directory.EnumerateFiles(@"C:\temp").GetEnumerator();
    try
    {
        while (e.MoveNext())
        {
            string file = e.Current;
            Console.WriteLine(file);
        }
    }
    finally
    {
        if (e != null)
        {
            ((IDisposable) e).Dispose();
        }
    }
}
```

В зависимости от типа перечислителя коллекции есть несколько вариантов, которые может создать компилятор. Если это значимый тип, реализующий `IDisposable`, компилятор не будет генерировать проверку на `null` в блоке `finally` (так же, как и в операторе `using`). Если статический тип перечислителя не реализует `IDisposable`, результат зависит от того, открыт ли тип для наследования. Если он запечатан или является значимым типом, компиля-

тор вообще не будет генерировать код, который пытается вызвать `Dispose`. Если он не запечатан, компилятор генерирует код в блоке `finally`, который во время выполнения проверяет, реализует ли перечислитель `IDisposable`. Если это так, вызывается `Dispose`; в противном случае он не делает ничего.

Интерфейс `IDisposable` проще всего использовать, когда вы получаете ресурс и в том же методе заканчиваете его использование, так как вы можете написать оператор `using` (или, если это возможно, цикл `foreach`), чтобы убедиться, что вы вызываете `Dispose`. Но иногда придется писать класс, который создает удаляемый объект и помещает ссылку на него в поле, потому что ему требуется использовать этот объект в течение более длительного времени. Например, можно написать класс журналирования, а если диспетчер журналирования записывает данные в файл, то ему может постоянно требоваться объект `StreamWriter`. В данном случае C# не облегчит вам жизнь, поэтому следует убедиться, что все содержащиеся объекты удалены. Придется написать свою собственную реализацию `IDisposable`, которая удаляет другие объекты. Как показывает листинг 7.16, это не бином Ньютона. Обратите внимание, что в этом примере для `_file` устанавливается значение `null`, поэтому не будет попыток утилизировать файл дважды. Это не является строго необходимым, потому что `StreamWriter` способен перенести многократные вызовы `Dispose`. Но он предоставляет объекту `Logger` простой способ узнать, что он находится в удаленном состоянии, поэтому, если бы мы добавили несколько реальных методов, мы могли бы проверять `_file` и вызвать исключение `ObjectDisposedException`, если эта переменная равна `null`.

Листинг 7.16. Освобождение содержащегося экземпляра

```
public sealed class Logger : IDisposable
{
    private StreamWriter _file;

    public Logger(string filePath)
    {
        _file = File.CreateText(filePath);
    }

    public void Dispose()
    {
        if (_file != null)
        {
            _file.Dispose();
            _file = null;
        }
    }
}
```

```
        }
    }
    // Конечно, настоящий класс продолжал бы что-то делать с StreamWriter.
}
```

Этот пример избегает серьезной проблемы. Класс запечатан, что позволяет снять вопрос о том, как быть с наследованием. Если вы пишете незапечатанный класс, который реализует `IDisposable`, стоит предоставить производному классу способ добавить свою собственную логику удаления. Наиболее простым решением было бы сделать `Dispose` виртуальным, чтобы производный класс мог переопределить его, выполняя собственную очистку в дополнение к вызову базовой реализации. Тем не менее есть немного более сложная схема, которая будет время от времени вам встречаться в .NET.

Некоторые объекты реализуют `IDisposable`, имея при этом финализатор. Начиная с введения `SafeHandle` и связанных классов, довольно необычно для класса иметь и то и другое (если он не наследуется от `SafeHandle`). Как правило, в финализации нуждаются только оболочки для дескрипторов, и классы, которые используют дескрипторы, теперь обычно используют `SafeHandle` вместо реализации своих собственных финализаторов. Однако существуют исключения, и в некоторых типах библиотек реализуется схема, разработанная для поддержки как финализации, так и `IDisposable`, что позволяет вам в производных классах определять собственное поведение для обоих случаев. Например, так работает базовый класс `Stream`.



Эта схема называется `Dispose`-паттерн, но не следует думать, что вам всегда требуется использовать ее при реализации `IDisposable`. Наоборот, весьма нетипично, если вам нужен этот шаблон. В нем нуждалось не так много классов даже на момент его появления, а теперь, когда есть `SafeHandle`, необходимость в нем практически отпала. (`SafeHandle` появился в .NET 2.0, и время, когда этот шаблон был полезен, уже давно ушло.) К сожалению, некоторые люди неправильно поняли его узкую полезность, поэтому вы периодически будете натыкаться на совершенно бесполезные советы о том, что следует использовать его для всех реализаций `IDisposable`. Игнорируйте их. Главное, что сегодня следует о нем знать, — это то, что он иногда встречается в старых типах вроде `Stream`.

Шаблон определяет защищенную перегрузку `Dispose`, которая принимает один аргумент `bool`. Базовый класс вызывает ее из своего открытого метода

`Dispose`, а также из своего деструктора, передавая `true` или `false` соответственно. Таким образом, вы должны переопределить только один метод, `protected Dispose`. Он может содержать логику, общую как для финализации, так и для удаления, например закрытие дескрипторов, но вы также можете выполнять любые действия, специфичные для удаления или финализации, потому что аргумент сообщает вам, какой тип очистки выполняется. Листинг 7.17 показывает, как это может выглядеть.

Этот гипотетический пример является пользовательской реализацией абстракции `Stream`, в которой используется некая внешняя библиотека, отличная от .NET, обеспечивающая доступ к ресурсам на основе дескриптора. Предпочтительно закрывать дескриптор при вызове открытого метода `Dispose`, но если этого не произошло к моменту запуска нашего финализатора, нужно закрыть дескриптор тогда. Таким образом, код проверяет, открыт ли дескриптор, и закрывает его при необходимости. Он делает это независимо от того, произошел ли вызов перегрузки `Dispose (bool)` в результате явного удаления объекта или его завершения, — мы должны убедиться, что дескриптор закрыт в любом случае. Однако этот класс также использует экземпляр класса `Logger` из листинга 7.16. Поскольку это обычный объект, нам не следует пытаться использовать его во время финализации, так что мы пытаемся освободить его, только если удаляется наш объект. Если происходит финализация, то, хотя сам `Logger` не является финализируемым, он использует `FileStream`, который является финализуемым; и вполне возможно, что финализатор `FileStream` уже будет запущен ко времени запуска финализатора нашего класса `MyFunkyStream`, поэтому вызов методов класса `Logger` — плохая идея.

Листинг 7.17. Собственная логика финализации и удаления

```
public class MyFunkyStream : Stream
{
    // Только для примера. Лучше избегать всего этого
    // шаблона и использовать вместо него тип, наследуемый из SafeHandle.
    private IntPtr _myCustomLibraryHandle;
    private Logger _log;

    protected override void Dispose(bool disposing)
    {
        base.Dispose(disposing);

        if (_myCustomLibraryHandle != IntPtr.Zero)
        {
```

```
        MyCustomLibraryInteropWrapper.Close(_myCustomLibraryHandle);
        _myCustomLibraryHandle = IntPtr.Zero;
    }
    if (disposing)
    {
        if (_log != null)
        {
            _log.Dispose();
            _log = null;
        }
    }
}
// ... перегрузки абстрактных методов Stream попали бы сюда
}
```

Когда базовый класс содержит этот виртуальный защищенный вариант `Dispose`, он должен вызвать `GC.SuppressFinalization` в своем публичном `Dispose`.

Базовый класс `Stream` делает это. В более общем плане, если вы пишете класс, который содержит и `Dispose`, и финализатор, то независимо от того, желаете ли вы в своей схеме поддерживать наследование, вам следует в любом случае подавить финализацию при вызове `Dispose`.

Необязательное удаление

Хотя вы должны вызывать `Dispose` в большинстве случаев для большинства объектов, реализующих `IDisposable`, есть несколько исключений. Например, `Reactive Extensions` для .NET (описываются в главе 11) содержат объекты `IDisposable`, которые позволяют подписываться на потоки событий. Вы можете вызвать `Dispose`, чтобы отписаться, но некоторые источники событий завершаются естественным образом, автоматически отключая любые подписки. Если это происходит, не обязательно вызывать `Dispose`. Кроме того, тип `Task`, который широко используется в сочетании с методами асинхронного программирования, описанными в главе 17, реализует `IDisposable`, но его не нужно удалять, пока вы не заставите его выделить `WaitHandle`, чего не произойдет при обычном использовании. Обычное использование `Task` делает затруднительным поиск подходящего времени для вызова `Dispose`, так что хорошо, что в этом нет необходимости.

Класс `HttpClient` — это еще одно исключение из обычных правил, работающее по-другому. `Dispose` редко вызывается для экземпляров этого типа, но

в данном случае благодаря тому, что мы поощряем повторное использование экземпляров. Если вы создаете, используете и удаляете `HttpClient` каждый раз, когда он вам нужен, вы лишаете его возможности повторно использовать существующие соединения при отправке нескольких запросов на один и тот же сервер. Это может вызвать две проблемы. Во-первых, открытие HTTP-соединения иногда может занимать больше времени, чем отправка запроса и получение ответа, поэтому предотвращение повторного использования `HttpClient` соединений для отправки нескольких запросов со временем может привести к заметным проблемам с производительностью. Повторное использование подключения работает только при повторном использовании `HttpClient`. Во-вторых, свойства протокола TCP (который лежит в основе HTTP) означают, что ОС не всегда может мгновенно вернуть все ресурсы, связанные с соединением: может потребоваться на значительное время забронировать TCP-порт соединения (возможно, на несколько минут) после того, как вы попросили ОС закрыть соединение. Это может привести к тому, что закончатся порты, а это предотвратит все дальнейшие соединения⁷.

Такие исключения необычны. Безопасно пропускать вызовы `Dispose` только в том случае, если в документации для используемого класса явно указано, что эти вызовы необязательны.

Упаковка

Говоря о сборке мусора и времени жизни объекта, я должен затронуть еще одну тему, а именно *упаковку* (boxing). Упаковка — это процесс, который позволяет переменной типа `object` ссылаться на значимый тип. Если переменная типа `object` способна содержать только ссылку на объект в куче, как же она может ссылаться на `int`? Что произойдет, если запустить код в листинге 7.18?

Листинг 7.18. Использование `int` в качестве объекта

```
class Program
{
    static void Show(object o)
    {
        Console.WriteLine(o.ToString());
```

⁷ Стого говоря, повторно использовать нужно `MessageHandler`. Если вы получаете `HttpClient` из `IHttpClientFactory`, утилизировать его безопасно, потому что фабрика сохраняет обработчик и повторно использует его в экземплярах `HttpClient`.

```
}

static void Main(string[] args)
{
    int num = 42;
    Show(num);
}
```

Метод `Show` ожидает объект, но я передаю ему `num`, который является локальной переменной значимого типа `int`. В этих обстоятельствах C# генерирует блок, который по сути является оболочкой ссылочного типа для значения. CLR может автоматически предоставлять упаковку для любого значимого типа, хотя если она этого не делает, то вы можете написать собственный класс с таким функционалом. Листинг 7.19 показывает упаковку вручную.

Листинг 7.19. Не совсем так, как работает упаковка

```
// Не настоящая упаковка, но эффект схожий.
public class Box<T>
    where T : struct
{
    public readonly T Value;
    public Box(T v)
    {
        Value = v;
    }
    public override string ToString() => Value.ToString();
    public override bool Equals(object obj) => Value.Equals(obj);
    public override int GetHashCode() => Value.GetHashCode();
}
```

Это довольно обычный класс, который содержит единственный экземпляр значимого типа в качестве единственного поля. Если вы вызываете стандартные члены `object` применительно к упаковке, переопределения этого класса заставляют это выглядеть так, как будто вы вызывали их непосредственно для самого поля. Следовательно, если я передал `new Box<int>(num)` в качестве аргумента для `Show` в листинге 7.18, `Show` получит ссылку на эту упаковку. Когда `Show` вызовет `ToString`, упаковка вызовет `ToString` поля `int`, поэтому можно ожидать, что программа отобразит 42.

Нам не нужно писать листинг 7.19 самостоятельно, потому что CLR создает упаковку за нас. Она создаст объект в куче, который содержит копию

упакованного значения, и перенаправляет стандартные методы `object` в упакованное значение. И делает еще кое-что, чего мы не можем. Если вы запросите у упакованного `int` его тип с помощью `GetType`, он вернет тот же объект `Type`, который вы получили бы, если бы вызывали `GetType` непосредственно для переменной `int`. Я не могу проделать это с моим собственным `Box<T>`, потому что `GetType` не является виртуальным. Кроме того, получить базовое значение проще, чем при ручном методе, поскольку распаковка является встроенной функцией CLR.

Если у вас есть ссылка на тип `object`, которую вы приводите к типу `int`, CLR проверяет, действительно ли ссылка указывает на упакованный тип `int`; если это так, CLR возвращает копию упакованного значения. (Если нет, то генерируется исключение `InvalidCastException`.) Таким образом, внутри метода `Show` листинга 7.18 я мог бы написать `(int) o`, чтобы получить копию исходного значения, тогда как при использовании класса из листинга 7.19, мне бы понадобилось более запутанное `((Box<int>) o).Value`.

Для извлечения упакованного значения я также могу использовать сопоставление с шаблоном. В листинге 7.20 используется шаблон типа, чтобы определить, содержит ли переменная `o` ссылку на упакованный `int`, и, если это так, извлекает его в локальную переменную `i`. Как мы видели в главе 2, когда вы так используете шаблон с оператором `is`, результирующее выражение вычисляется как `true`, если шаблон совпадает, и `false`, если нет. Таким образом, тело этого оператора `if` выполняется только в том случае, если там содержится значение `int`, которое необходимо распаковать.

Листинг 7.20. Распаковка значения шаблоном типа

```
if (o is int i)
{
    Console.WriteLine(i * 2);
}
```

Упаковка автоматически доступна для всех структур, а не только для встроенных значимых типов⁸. Если структура реализует какие-либо интерфейсы, упаковка предоставит все те же интерфейсы. (Это еще один фокус, который код из листинга 7.19 не может выполнить.)

Некоторые неявные преобразования вызывают упаковку. Вы можете наблюдать такое в листинге 7.18. Я передал выражение типа `int` туда, где

⁸ За исключением типов `ref struct`, потому что они всегда остаются в стеке.

требуется `object`, без необходимости явного приведения. Неявные преобразования также возможны между значением и любым из интерфейсов, которые реализует значимый тип. Например, можно присвоить значение типа `int` переменной типа `IComparable<int>` (или передать его как аргумент этого типа в метод) без необходимости приведения. Это приводит к созданию упаковки, потому что переменные любого типа интерфейса похожи на переменные типа `object`, поскольку могут содержать только ссылку на элемент в куче, обрабатываемой сборщиком мусора.

Неявная упаковка может иногда вызывать проблемы по одной из двух причин. Во-первых, с ее помощью легко добавить работы сборщику мусора. CLR не пытается кэшировать упаковки, поэтому, если вы напишите цикл, который выполняется 100 000 раз, и этот цикл содержит выражение, использующее неявное преобразование в упаковки, вы в конечном итоге создадите 100 000 упаковок, которые сборщику мусора когда-то придется освобождать, как и все остальное в куче. Во-вторых, каждая операция упаковки (и каждая распаковка) копирует значение, что может не дать ожидаемой семантики. Листинг 7.21 иллюстрирует потенциально неожиданное поведение.

Листинг 7.21. Ловушки изменяемых структур

```
public struct DisposableValue : IDisposable
{
    private bool _disposedYet;

    public void Dispose()
    {
        if (!_disposedYet)
        {
            Console.WriteLine("Disposing for first time");
            _disposedYet = true;
        }
        else
        {
            Console.WriteLine("Was already disposed");
        }
    }

    class Program
    {
```

```
static void CallDispose(IDisposable o)
{
    o.Dispose();
}

static void Main(string[] args)
{
    var dv = new DisposableValue();
    Console.WriteLine("Passing value variable:");
    CallDispose(dv);
    CallDispose(dv);
    CallDispose(dv);

    IDisposable id = dv;
    Console.WriteLine("Passing interface variable:");
    CallDispose(id);
    CallDispose(id);
    CallDispose(id);

    Console.WriteLine("Calling Dispose directly on value variable:");
    dv.Dispose();
    dv.Dispose();
    dv.Dispose();

    Console.WriteLine("Passing value variable:");
    CallDispose(dv);
    CallDispose(dv);
    CallDispose(dv);
}
```



Неявные преобразования в упаковку не являются неявными ссылочными преобразованиями. Это означает, что они не оказывают влияния на ковариантность или контравариантность. Например, `IEnumerable<int>` не совместим с `IEnumerable<object>`, несмотря на существование неявного преобразования из `int` в `object`, потому что это не неявное преобразование ссылки.

Структура `DisposableValue` реализует интерфейс `IDisposable`, который мы уже встречали ранее. Он отслеживает, был ли объект уже удален. Программа содержит метод `CallDispose`, который вызывает `Dispose` для любого экземпляра `IDisposable`. Она объявляет одну переменную типа `DisposableValue` и трижды передает ее в `CallDispose`. Вот вывод этой части программы:

```
Passing value variable:  
Disposing for first time  
Disposing for first time  
Disposing for first time
```

Во всех трех случаях структура, похоже, считает, что это ее первый вызов `Dispose`. Это связано с тем, что каждый вызов `CallDispose` создавал новую упаковку. На самом деле мы не передаем переменную `dv`, а каждый раз передаем новую упакованную копию, поэтому метод `CallDispose` каждый раз работает с другим экземпляром структуры. Это согласуется с тем, как обычно работают значимые типы — даже когда они не упакованы, то, передавая их в качестве аргументов, вы в конечном итоге передаете копию (если не используете ключевые слова `ref` и `in`).

Следующая часть программы генерирует только одну упаковку — она присваивает значение другой локальной переменной типа `IDisposable`. При этом используется то же неявное преобразование, что и при передаче переменной непосредственно в качестве аргумента, поэтому создается еще одна упаковка, но делается это только один раз. Затем мы трижды передаем одну и ту же ссылку на эту конкретную упаковку, что объясняет, почему выходные данные этой фазы программы выглядят по-другому:

```
Passing interface variable:  
Disposing for first time  
Was already disposed  
Was already disposed
```

Все эти три вызова `CallDispose` используют одну и ту же упаковку, которая содержит экземпляр нашей структуры, и поэтому после первого вызова объект запоминает, что он уже был удален. Затем наша программа вызывает `Dispose` непосредственно для локальной переменной, давая такой вывод:

```
Calling Dispose directly on value variable:  
Disposing for first time  
Was already disposed  
Was already disposed
```

Здесь не используется никакой упаковки, поэтому мы модифицируем состояние локальной переменной. Тот, кто лишь бегло взглянул на код, возможно, не будет ожидать такого вывода — мы уже передали переменную `dv` методу, который вызвал `Dispose` в своем аргументе, поэтому может показаться удивительным то, что он считает, что не был удален в первый раз. Но, как только вы поймете, что `CallDispose` требует ссылки и, следовательно,

не может использовать значение напрямую, становится ясно, что каждый вызов `Dispose` до этой точки работал с какой-то упакованной копией, а не с локальной переменной.

Наконец, мы делаем еще три вызова, снова передавая `dv` напрямую в `Call-Dispose`. Это именно то, что мы сделали в начале кода, поэтому вызовы генерируют еще больше упакованных копий. Но на этот раз мы копируем значение, которое уже находится в состоянии удаления, поэтому видим другой результат:

```
Passing value variable:  
Was already disposed  
Was already disposed  
Was already disposed
```

Когда вы понимаете, что происходит, поведение становится простым, но требует от вас помнить, что вы имеете дело с типом значения, и понимать, когда упаковка вызывает неявное копирование. Это одна из причин, по которой Microsoft не рекомендует разработчикам писать значимые типы, которые могут изменить свое состояние, — если значение не может измениться, то упакованное значение этого типа также не может измениться. Труднее запутаться, когда не имеет значения, работаете ли вы с оригиналом или с упакованной копией. Однако все же полезно понимать, когда происходит упаковка, чтобы избежать снижения производительности.

В ранних версиях .NET упаковка была более распространенным явлением. До того как в .NET 2.0 появились обобщения, все классы коллекций работали в терминах `object`, поэтому, если вы хотели получить список целых чисел с изменяемым размером, все кончалось упаковкой каждого `int` из списка. Обобщенные классы коллекции не вызывают упаковку — `List<int>` способен хранить распакованные значения напрямую.

Упаковка `Nullable<T>`

В главе 3 был описан тип `Nullable<T>`, оболочка, которая добавляет поддержку значения `null` любому значимому типу. Как вы помните, C# содержит для этого специальный синтаксис, когда ставится вопросительный знак в конце имени значимого типа. Поэтому обычно мы пишем `int?` вместо `Nullable<int>`. CLR специальным образом поддерживает упаковку `Nullable<T>`.

`Nullable<T>` сам по себе является значимым типом, поэтому, если вы попытаетесь получить ссылку на него, компилятор генерирует код, который

пытается его упаковать, как это было бы с любым другим значимым типом. Однако во время выполнения CLR не будет создавать упаковку, содержащую копию `Nullable<T>`. Вместо этого среда проверяет, находится ли значение в состоянии `null` (т. е. его свойство `HasValue` возвращает `false`), и, если да, просто возвращает `null`. В противном случае она упаковывает содержащееся значение. Например, если `Nullable<int>` содержит значение, будет произведена упаковка типа `int`. Это будет выглядеть неотличимо от упаковки, которую бы вы получили, если бы делали ее для обычного значения `int`. (Одним из результатов является то, что сопоставление с шаблоном, показанное в листинге 7.20, работает независимо от того, был ли тип изначально упакованной переменной типом `int` или `int?`. В шаблоне типа в любом случае используется `int`.)

Вы можете распаковать упакованный `int` в переменные как типа `int?`, так и `int`. Таким образом, все три операции распаковки в листинге 7.22 будут успешными. Они также были бы успешными, если бы первая строка была изменена с целью инициализации переменной `boxed` из `Nullable<int>`, которая находится не в нулевом состоянии. (Если бы вы инициализировали `boxed` из `Nullable <int>` в состоянии `null`, это бы имело тот же эффект, что и инициализация ее в `null`. В этом случае последняя строка примера выдала бы исключение `NullReferenceException`.)

Листинг 7.22. Распаковка `int` в переменные, допускающие и не допускающие значение `null`

```
object boxed = 42;
int? nv = boxed as int?;
int? nv2 = (int?) boxed;
int v = (int) boxed;
```

Это функция времени выполнения, а не показатель разумности компилятора. Инструкция промежуточного языка `box`, которая генерируется C# при упаковке значения, обнаруживает значение `Nullable<T>`; инструкции промежуточного языка `unbox` и `unbox.any` могут производить значение `Nullable<T>` как из `null`, так и из ссылки на упакованное значение базового типа. Таким образом, если вы напишете собственный тип оболочки, который выглядит как `Nullable<T>`, он не будет вести себя так же; если бы вы присвоили `object` значение вашего типа, он упаковал бы всю вашу обертку, как и любое другое значение. Лишь только благодаря поддержке CLR типа `Nullable<T>` он ведет себя по-другому.

Итог

В этой главе я описал кучу, которую предоставляет среда выполнения. Я продемонстрировал стратегию, которую CLR использует для определения того, какие объекты кучи все еще достижимы для вашего кода, а также механизм на основе поколений, который используется для восстановления памяти, занятой больше не используемыми объектами. Сборщик мусора не ясновидящий, поэтому, если ваша программа поддерживает достижимость объекта, он предполагает, что вы хотите использовать этот объект в будущем. Это означает, что вам иногда стоит проявлять осторожность, чтобы избежать утечек памяти, случайно удерживая объекты слишком долго. Мы рассмотрели механизм финализации, его ограничения и проблемы с производительностью, а также `IDisposable`, который является предпочтительным способом для очистки ресурсов, не являющихся памятью. Наконец, мы увидели, как благодаря упаковке значимые типы могут работать как ссылочные.

В следующей главе я покажу, как C# дает возможность использовать механизмы обработки ошибок CLR.

ГЛАВА 8

Исключения

Некоторые операции могут завершиться неудачей. Если ваша программа читает данные из файла, хранящегося на внешнем диске, кто-то может отключить диск. Приложение может попытаться создать массив и обнаружить, что в системе недостаточно свободной памяти. Ненадежное подключение к беспроводной сети может привести к сбою в работе сетевых запросов. Один из широко используемых способов обнаружения подобного рода ошибок в API — возвращать значение, указывающее, было ли выполненное действие успешным. Это требует бдительности от разработчиков относительно того, все ли ошибки они отслеживают, потому что программе нужно проверять возвращаемое значение каждой операции. Это, безусловно, жизнеспособная стратегия, но она может запутать код; логическая последовательность задач, выполняемых в нормальных условиях, может оказаться похороненной под всеми этими проверками на ошибки, что усложняет сопровождение кода. C# поддерживает другой популярный механизм обработки ошибок, который способен сгладить эту проблему, — *исключения*.

Когда API сообщает об ошибке с исключением, это нарушает нормальный ход выполнения, что приводит к переходу к ближайшему подходящему коду обработки ошибок. Механизм обеспечивает определенный уровень разделения между логикой обработки ошибок и кодом, который пытается выполнить поставленную задачу. Он может облегчить чтение и обслуживание кода, хотя, с другой стороны, способен затруднить просмотр всех возможных путей выполнения кода.

Исключения также могут сообщать о проблемах в операциях в тех случаях, когда код возврата оказывается неприменим. Например, среда выполнения может обнаруживать и сообщать о проблемах основных операций, даже таких простых, как использование ссылки. Переменные ссылочного типа могут содержать `null`, и, если вы попытаетесь вызвать метод с такой ссылкой, произойдет сбой. Среда выполнения сообщает об этом исключением.

Большинство ошибок в .NET представлены в виде исключений. Однако некоторые API предлагают выбор между кодами возврата и исключениями. Например, тип `int` имеет метод `Parse`, принимающий строку и пытающийся интерпретировать ее содержимое как число. Если вы передадите ему какой-то нечисловой текст (например, «Hello»), он укажет на ошибку, вызвав исключение `FormatException`. Если вас это не устраивает, то вместо этого можно вызывать `TryParse`, который выполняет точно такую же работу, но если ввод не числовой, он возвращает `false` вместо вызова исключения. (Поскольку возвращаемое значение метода призвано сообщать об успехе или неудаче, метод предоставляет целочисленный результат через выходной параметр.) Числовой синтаксический анализ — не единственная операция, использующая эту схему, при которой используется пара методов (`Parse` и `TryParse`, в данном случае) и предоставляется выбор между исключениями и возвращаемыми значениями. Как вы видели в главе 5, аналогичный выбор предоставляют словари. Индексатор выдает исключение, если вы используете ключ, которого нет в словаре, но вы также можете искать значения с помощью `TryGetValue`, который возвращает `false` при сбое, подобно `TryParse`. Хотя эта схема и встречается в ряде мест, для большинства API исключения остаются единственным выбором.

Если вы разрабатываете API, который способен вызвать сбой, как он должен об этом сбое сообщать? Использовать ли в этом случае исключения, возвращаемое значение или и то и другое? Рекомендации Microsoft по разработке библиотеки классов содержат инструкции, которые кажутся однозначными:

Не возвращайте коды ошибок. Исключения являются основным средством сообщения об ошибках в библиотеках классов.

Рекомендации по разработке .NET Framework

Но как это согласуется с существованием `int.TryParse`? В руководстве есть раздел, посвященный вопросам производительности исключений, в котором говорится следующее:

Рассмотрите использование TryParse для элементов, которые могут генерировать исключения в общих сценариях, чтобы избежать проблем с производительностью, связанных с исключениями.

Рекомендации по разработке .NET Framework

Невозможность проанализировать число не обязательно является ошибкой. Например, нужно, чтобы ваше приложение позволяло указывать месяц как в виде числа, так и в виде текста. Так что, безусловно, есть распространенные сценарии, в которых операция может завершиться неудачей, но у руководства есть и другой критерий: предлагается использовать его для «чрезвычайно чувствительных к производительности API». Так что предлагать подход `TryParse` следует только тогда, когда операция выполняется быстро по сравнению со временем, необходимым для вызова и обработки исключения.

Исключения, как правило, могут создаваться и обрабатываться за доли миллисекунды, поэтому они не такие уж и медленные — во всяком случае, не такие, как чтение данных по сетевому соединению, — но и не слишком быстрые. Я выяснил, что на моем компьютере один поток может анализировать пятизначные числовые строки со скоростью примерно 65 миллионов строк в секунду в .NET Core 3.0 и он способен отклонять нечисловые строки с такой же скоростью при использовании `TryParse`. Метод `Parse` обрабатывает числовые строки так же быстро, но он примерно в 1000 раз медленнее отклоняет нечисловые строки, чем `TryParse`, благодаря затратам на исключения. Конечно, преобразование строк в целые числа — это довольно быстрая операция, поэтому это делает исключения неудачным выбором, но именно поэтому эта схема наиболее распространена для операций, быстрых по своей природе.



Особенно медленно исключения могут работать при отладке. Отчасти это связано с тем, что отладчику необходимо решить, куда ему влезать, но это особенно заметно при первом необработанном исключении, которое вызывает ваша программа. Может создаться впечатление, что исключения значительно дороже, чем они есть на самом деле. Числа в предыдущем абзаце основаны на наблюдаемом поведении во время выполнения без отладки. Тем не менее эти цифры несколько занижают затраты, поскольку обработка исключения приводит к тому, что CLR запускает кусочки кода и получает доступ к структурам данных, которые в противном случае не нужны, а это может привести к вытеснению полезных данных из кэша ЦП. Это может привести к тому, что код будет работать медленнее в течение короткого времени после обработки исключения, пока не относящийся к исключению код и данные не вернутся в кэш. Простота теста уменьшает этот эффект.

Большинство API не предоставляют вариант `TryXXX` и сообщают обо всех сбоях как об исключениях, даже в случаях, когда сбой может быть вполне ожидаемым. Например, файловые API не содержат способа открыть для чтения существующий файл без исключения в случае его отсутствия. (Вы можете сначала использовать другой API для проверки наличия файла, но это не гарантия успеха. Другой процесс всегда может удалить файл между вашим запросом касательно его существования и попыткой его открыть.) Поскольку операции с файловой системой по своей природе медленные, шаблон `TryXXX` не обеспечит здесь существенного повышения производительности, даже если добавит логики.

Источники исключений

API библиотеки классов — не единственный источник исключений. Они могут возникнуть в любом из следующих сценариев:

- Проблему обнаруживает ваш собственный код.
- Ваша программа использует API библиотеки классов, где возникает проблема.
- Среда выполнения обнаруживает сбой операции (например, арифметическое переполнение в проверяемом контексте, попытку использовать нулевую ссылку или разместить объект, для которого недостаточно памяти).
- Среда выполнения обнаруживает сбой вне вашего контроля, который влияет на ваш код (например, среда выполнения пытается выделить память для какой-то внутренней цели и обнаруживает, что свободной памяти недостаточно).

Хотя все они используют одни и те же механизмы обработки исключений, места возникновения исключений отличаются. Когда ваш собственный код вызовет исключение (позже я покажу вам, как это сделать), вы будете знать, какие условия привели к его возникновению, но что происходит, когда исключения вызывают другие сценарии? В следующих разделах я опишу, где ожидать каждый вид исключений.

Исключения от API

При вызове API есть несколько видов проблем, способных привести к исключениям. Возможно, вы предоставили аргументы, которые не имеют

смысла, например пустую ссылку вместо рабочей или пустую строку вместо имени файла. Или аргументы могут выглядеть хорошо по отдельности, но не все вместе. Например, вы можете вызвать API, который копирует данные в массив, и попросить его скопировать больше данных, чем в него умещается. Их можно описать как ошибки в стиле «никогда не сработает», и обычно они являются результатом ошибок в коде. (Один разработчик, который раньше работал в команде компилятора C#, называет их *тупоголовыми исключениями* (*boneheaded exceptions*).)

Другой класс проблем возникает, когда все аргументы выглядят правдоподобно, но операция оказывается невозможной при текущем состоянии среды. Например, вы можете попросить открыть определенный файл, но файл может отсутствовать; или, возможно, он существует, но какая-то другая программа уже открыла его и требует монопольного доступа. Еще один вариант заключается в том, что все может начаться хорошо, но поменяться в будущем. Например, вы успешно открыли файл и некоторое время читали данные, но затем файл стал недоступным. Как предлагалось ранее, кто-то мог отключить диск или диск мог выйти из строя из-за перегрева или возраста.

Программное обеспечение, которое обменивается данными с внешними службами по сети, должно учитывать, что исключение не обязательно указывает, что что-то действительно не так — иногда запросы не выполняются из-за какого-то временного обстоятельства, и просто требуется повторить операцию. Это особенно распространено в облачных средах, где отдельные серверы обычно включаются и выключаются в рамках балансировки нагрузки, которую, как правило, предлагают облачные платформы. Это нормально, когда несколько операций не дают результатов по какой-либо конкретной причине.



При использовании служб через библиотеку вы должны выяснить, обрабатывает ли она это за вас. Например, библиотеки хранилища Azure по умолчанию выполняют повторные попытки и выдают исключение только в том случае, если вы отключите это поведение или если проблемы не исчезнут после ряда попыток. Как правило, не нужно добавлять собственные обработки исключений и повторные циклы для такого рода ошибок в случае библиотек, которые делают это для вас.

Асинхронное программирование добавляет еще один вариант. В главах 16 и 17 я покажу различные асинхронные API, в которых работа может про-

должаться после возвращения из метода, который ее запустил. Работа, которая выполняется асинхронно, также асинхронно сбоят, и в этом случае библиотеке может потребоваться дождаться следующего вызова вашего кода, прежде чем она сможет сообщить об ошибке.

Несмотря на различия, во всех этих случаях исключение будет исходить из некоторого API, который вызывается вашим кодом. (Даже в случае сбоя асинхронных операций исключения возникают либо при попытке получить результат операции, либо когда вы явно спрашиваете, была ли ошибка.) В листинге 8.1 показан код, в котором могут возникать исключения такого рода.

Листинг 8.1. Получение исключения из библиотечного вызова

```
static void Main(string[] args)
{
    using (var r = new StreamReader(@"C:\Temp\File.txt"))
    {
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
```

В этой программе нет ничего категорически неправильного, поэтому мы не получим никаких исключений относительно того, что аргументы изначально неверные. (В неофициальной терминологии он не допускает тупоголовых ошибок.) Если на диске C: вашего компьютера есть папка Temp, и если она содержит файл File.txt, и если пользователь, запускающий программу, имеет разрешение на чтение файла, и если на компьютере никто не получил монопольного доступа к файлу, и если нет проблем — таких, как повреждение диска, — которые могут сделать любую часть файла недоступной, и если нет новых проблем (таких, как воспламенение диска) во время работы программы, этот код будет работать замечательно: он покажет каждую строку текста в файле. Но тут очень много *если*.

Если такого файла нет, конструктор `StreamReader` не будет завершен. Вместо этого он выдаст исключение. Эта программа не пытается обработать его, поэтому приложение будет завершено. Если вы запустили программу за пределами отладчика Visual Studio, то увидите следующий вывод:

```
Unhandled Exception: System.IO.DirectoryNotFoundException: Could not
find a part of the path 'C:\Temp\File.txt'.
   at System.IO.FileStream.ValidateFileHandle(SafeFileHandle
fileHandle)
   at System.IO.FileStream.CreateFileOpenHandle(FileMode mode,
FileShare share, FileOptions options)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess
access, FileShare share, Int32 bufferSize, FileOptions options)
   at System.IO.StreamReader.ValidateArgsAndOpenPath(String path,
Encoding encoding, Int32 bufferSize)
   at System.IO.StreamReader..ctor(String path)
   at Exceptional.Program.Main(String[] args) in c:\Examples\Ch08\
Example1\Program.cs:line 10
```

Он сообщает нам, какая ошибка произошла, и показывает полный стек вызовов программы в момент возникновения проблемы. В Windows также включится общесистемная обработка ошибок, поэтому в зависимости от конфигурации вашего компьютера вы увидите диалоговое окно отчетов об ошибках и даже сможете сообщить о сбое в Microsoft. Если вы запустите ту же программу в отладчике, она сообщит вам об исключении, а также выделит строку, в которой произошла ошибка, как показано на рис. 8.1.

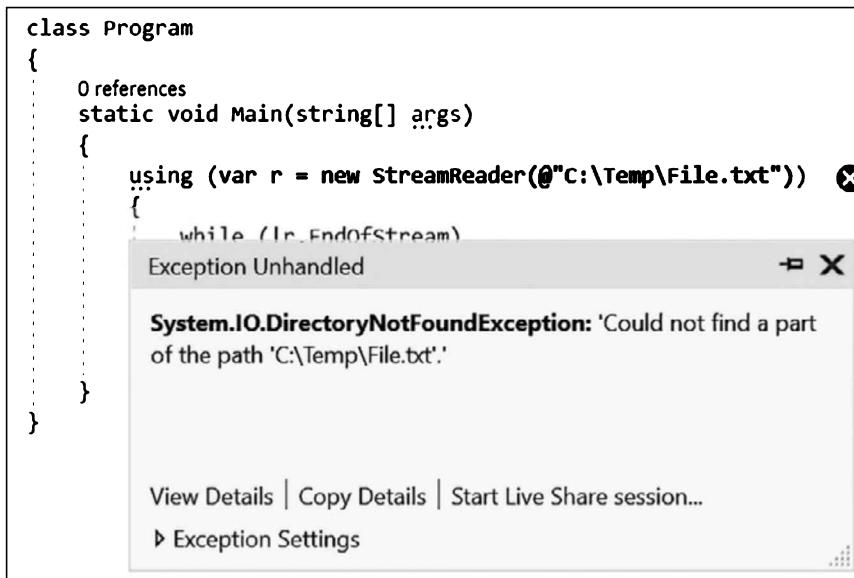


Рис. 8.1. Visual Studio сообщает об исключении

Здесь мы видим поведение по умолчанию, которое возникает, когда программа ничего не делает для обработки исключений: если подключен отладчик, он включается, а если нет — программа просто аварийно завершается. Очень скоро я покажу, как обрабатывать исключения, но пример показывает, что вы не можете просто игнорировать их.

Между прочим, вызов конструктора `StreamReader` — не единственная строка, которая может вызвать исключение в листинге 8.1. Код вызывает `ReadLine` несколько раз, и любой из этих вызовов может завершиться ошибкой. В целом любой доступ к элементу может привести к исключению, включая простое чтение свойства, хотя разработчики библиотеки классов обычно пытаются ограничить масштабы, в которых свойства генерируют исключения. Если вы сделаете ошибку типа «это никогда не сработает» (тупоголовую), свойство может вызвать исключение, но такого обычно не делается для ошибок типа «эта конкретная операция не сработала». Например, в документации сказано, что свойство `EndOfStream` из листинга 8.1 выдает исключение при попытке прочитать его после вызова `Dispose` объекта `StreamReader`, что является очевидной ошибкой кодирования. Но если имеются проблемы с чтением файла, `StreamReader` будет генерировать исключения только из методов или конструктора.

Сбои, обнаруженные во время выполнения

Еще один источник исключений — это когда CLR сама обнаруживает, что некая операция завершилась неудачно. Листинг 8.2 показывает метод, в котором это может произойти. Как и в листинге 8.1, в самом коде нет ничего плохого (кроме того, что он не особо полезен). Его вполне можно использовать безо всяких проблем. Но если кто-то передаст `0` в качестве второго аргумента, код попытается выполнить недопустимую операцию.

Листинг 8.2. Потенциальная ошибка во время выполнения

```
static int Divide(int x, int y)
{
    return x / y;
}
```

CLR обнаружит попытку деления на ноль и выдаст исключение `DivideByZeroException`. Это будет иметь тот же эффект, что и исключение из вызова API: если программа не попытается обработать исключение, произойдет сбой или в дело вступит отладчик.



В C# деление на ноль не всегда запрещено. Типы с плавающей точкой поддерживают специальные значения, представляющие положительную и отрицательную бесконечность, которые вы получаете, когда делите положительное или отрицательное значение на ноль; если вы разделите ноль на себя, вы получите специальное значение «Not a Number». Ни один из целочисленных типов не поддерживает эти специальные значения, поэтому целочисленное деление на ноль всегда является ошибкой.

Последним из описанных ранее источником исключений тоже является обнаружение определенных сбоев средой выполнения, но они работают немного по-другому. Они не обязательно напрямую инициируются тем, что ваш код делал в потоке, в котором происходит исключение. Иногда их называют асинхронными исключениями, и теоретически они могут быть сгенерированы буквально в любой точке вашего кода, что затрудняет обеспечение их правильной обработки. Тем не менее они, как правило, вызываются только в самых катастрофических условиях, часто перед закрытием вашей программы, поэтому вам не удастся с пользой их обработать. Например, в .NET Core, `StackOverflowException` и `OutOfMemoryException` теоретически могут быть выброшены в любой момент (поскольку CLR может потребоваться память для собственных целей, даже если ваш код не делал ничего, что явно к этому привело).

Я описал обычные ситуации, в которых генерируются исключения, и вы увидели поведение по умолчанию. Но что, если вы хотите, чтобы ваша программа выполняла что-то кроме аварийного завершения?

Обработка исключений

Когда возникает исключение, CLR ищет код для его обработки. Обработка исключений по умолчанию вступает в дело только в том случае, если во всем стеке вызовов не нашлось подходящих обработчиков. Для написания обработчика в C# мы используем ключевые слова `try` и `catch`, как показано в листинге 8.3.

Блок сразу за ключевым словом `try` обычно называется блоком `try`, и, если программа выдает исключение, находясь внутри такого блока, CLR ищет подходящие блоки `catch`. В листинге 8.3 есть только один блок `catch`, и из находящегося в скобках после ключевого слова `catch` видно, что этот конкретный блок предназначен для обработки исключений типа `FileNotFoundException`.

Листинг 8.3. Обработка исключения

```
try
{
    using (StreamReader r = new StreamReader(@"C:\Temp\File.txt"))
    {
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
catch (FileNotFoundException)
{
    Console.WriteLine("Couldn't find the file");
}
```

Ранее вы уже видели, что если нет файла C:\Temp\File.txt, то конструктор `StreamReader` выдает исключение `FileNotFoundException`. В листинге 8.1 это привело к аварийному завершению программы, но поскольку в листинге 8.3 имеется блок `catch` для этого исключения, CLR выполнит его. На этом этапе он будет считать, что исключение обработано, поэтому программа не завершает работу аварийно. Наш блок `catch` может делать все, что угодно, и в данном случае мой код просто отображает сообщение, указывающее, что файл отсутствует.

Обработчики исключений не обязательно располагаться в методе, в котором возникло исключение. CLR идет вверх по стеку, пока не найдет подходящий обработчик. Если бы вызывающий сбой вызов конструктора `StreamReader` был в каком-то другом методе, который был вызван из блока `try` в листинге 8.3, наш блок `catch` все равно сработал бы (если только этот метод не предоставил собственный обработчик для того же исключения).

Объекты исключений

Исключения являются объектами, а их тип является производным от базового класса `Exception`. Он определяет свойства, содержащие информацию об исключении, а некоторые производные типы добавляют свойства, специфичные для проблемы, о которой исключение сообщает¹. Ваш блок `catch`

¹ Стого говоря, CLR позволяет любому типу выступать в роли исключения. Тем не менее C# может выдавать только исключения типов, производных от `Exception`. Некоторые языки позволяют вам использовать другие типы, но это крайне нежела-

может получить ссылку на исключение, если ему нужна информация о том, что пошло не так. Листинг 8.4 показывает модификацию блока `catch` из листинга 8.3. В скобках после ключевого слова `catch` вместе с указанием типа исключения мы предоставляем идентификатор (`x`), с помощью которого код в блоке `catch` может ссылаться на объект исключения. Это позволяет коду прочитать свойство, специфичное для класса `FileNotFoundException`: `FileName`.

Листинг 8.4. Использование исключения в блоке `catch`

```
try
{
    // ... тот же код, что и в листинге 8.3 ...
}
catch (FileNotFoundException x)
{
    Console.WriteLine($"File '{x.FileName}' is missing");
}
```

Код отобразит имя файла, который отсутствует. С помощью этой простой программы мы уже узнали, какой именно файл мы пытались открыть, но вполне можно представить, как это свойство может пригодиться в более сложной программе, которая работает с несколькими файлами.

К элементам общего назначения, определенным базовым классом `Exception`, относится свойство `Message`, которое возвращает строку, содержащую текстовое описание проблемы. Его показывает обработка ошибок по умолчанию для консольных приложений. Текст `Could not find file 'C:\Temp\File.txt'`, который мы видели при первом запуске листинга 8.1, как раз из свойства `Message`. Это свойство важно, когда вы диагностируете неожиданные исключения.

`Exception` также определяет свойство `InnerException`. Оно часто содержит `null`, но вступает в дело, когда операция завершается неудачей в результате какой-то другой ошибки. Иногда исключения, возникающие глубоко внутри библиотеки, теряют смысл, если им разрешить распространяться вплоть до вызывающей стороны. Например, .NET предоставляет библиотеку для анализа файлов XAML. (XAML – расширяемый язык разметки

тельно. C# может обрабатывать исключения любого типа хотя бы потому, что компилятор автоматически устанавливает атрибут `RuntimeCompatibility` для каждого создаваемого им компонента, прося CLR оборачивать исключения, не полученные из `Exception`, в исключение `RuntimeWrappedException`.

приложений — используется различными платформами .NET UI, включая WPF.) XAML является расширяемым, поэтому возможно, что ваш код (или, возможно, какой-то сторонний код) будет выполняться как часть процесса загрузки XAML-файла, и этот код расширения может завершиться ошибкой. Предположим, что ошибка в вашем коде вызывает исключение `IndexOutOfRangeException` при попытке доступа к элементу массива. Было бы довольно загадочно, если бы это исключение выдал API XAML, поэтому независимо от основной причины сбоя библиотека выдает исключение `XamlParseException`. Это означает, что, если вы хотите обработать сбой при загрузке файла XAML, вы точно знаете, какое исключение следует обработать. Но основная причина сбоя тоже не теряется: когда сбой вызвало какое-то другое исключение, оно будет помещено в `InnerException`.



Свойство `Message` предназначено для использования человеком, поэтому многие API локализуют эти сообщения. Поэтому написание кода, который пытается интерпретировать исключение путем проверки свойства `Message`, — это плохая идея. Он может привести к сбою, если работает на компьютере, настроенном для работы в регионе, где основной язык отличается от вашего. (И Microsoft не рассматривает изменения сообщений об исключениях как критические изменения, поэтому текст может изменяться даже в пределах одной и той же локали.) Лучше полагаться на фактический тип исключения, хотя, как вы увидите в главе 15, некоторые исключения, такие как `IOException`, используются неоднозначными способами. Поэтому иногда вам придется обращаться к свойству `HResult`, которое в таких случаях будет содержать код ошибки от ОС.

Все исключения содержат информацию о месте своего возникновения. Свойство `StackTrace` содержит стек вызовов в виде строки. Как вы уже видели, обработчик исключений по умолчанию для консольных приложений его отображает. Также имеется свойство `TargetSite`, которое сообщает вам, какой метод выполнялся. Он возвращает экземпляр класса `MethodBase` API отражения. За подробностями касательно отражений обращайтесь к главе 13.

Несколько блоков `catch`

За блоком `try` могут следовать несколько блоков `catch`. Если первый `catch` не соответствует сгенерированному исключению, CLR обратится к следую-

щему, затем к следующему и т. д. В листинге 8.5 содержатся обработчики как для `FileNotFoundException`, так и для `IOException`.

Листинг 8.5. Обработка нескольких типов исключений

```
try
{
    using (StreamReader r = new StreamReader(@"C:\Temp\File.txt"))
    {
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
catch (FileNotFoundException x)
{
    Console.WriteLine($"File '{x.FileName}' is missing");
}
catch (IOException x)
{
    Console.WriteLine($"IO error: '{x.Message}'");
}
```

Интересной особенностью этого примера является то, что `FileNotFoundException` наследуется от `IOException`. Я мог бы удалить первый блок `catch`, и код все равно бы правильно обработал исключение (только с менее конкретным сообщением), потому что CLR считает блок `catch` соответствующим, если он обрабатывает базовый тип исключения. Таким образом, в листинге 8.5 есть два конкурентных обработчика для исключения `FileNotFoundException`, и в таких случаях C# требует, чтобы более конкретный следовал первым. Если бы я поменял их местами так, чтобы первым оказался обработчик `IOException`, я бы получил ошибку компилятора для обработчика `FileNotFoundException`:

```
error CS0160: A previous catch clause already catches all exceptions
of this or of a super type ('IOException')
```

Если вы напишете блок `catch` для базового типа `Exception`, он будет перехватывать вообще все исключения. В большинстве случаев так делать не следует. Если нет какого-то конкретного и полезного действия, которое вы можете проделать с исключением, обычно вам следует позволить ему пройти дальше. В противном случае вы рискуете скрыть проблему. Если вы дадите исключению пройти дальше, оно с большей вероятностью по-

падет туда, где будет замечено, что повысит шансы на то, что вы однажды исправите проблему. Обработчик всех исключений будет уместным, если вы намереваетесь обернуть все исключения в другое исключение и выбросить его, как описано выше в случае с `XamlParseException`. Обработчик всех исключений также может иметь смысл, если он находится там, где единственным оставшимся способом обработать исключение является обработка по умолчанию, предоставляемая системой. (Это может означать метод `Main` для консольного приложения, но для многопоточных приложений — код в верхней части стека вновь созданного потока.) В этих местах может оказаться целесообразным перехватывать все исключения и передавать подробности в лог-файл или какой-то похожий диагностический механизм. Но даже после регистрации, вы, вероятно, захотите повторно выбросить исключение, как описано далее в этой главе, или даже завершить процесс с ненулевым кодом завершения.



Для критически важных служб у вас может возникнуть соблазн написать код, который проглотит исключение, чтобы ваше приложение могло работать дальше. Это плохая идея. Если возникает исключение, которого вы не ожидали, внутреннее состояние вашего приложения может перестать быть надежным, поскольку ваш код мог лишь наполовину выполнить какую-то операцию, когда произошла ошибка. Если вы не можете позволить приложению отключиться, лучше всего организовать автоматический перезапуск приложения после сбоя. Например, служба Windows может быть настроена для этого автоматически.

Фильтры исключений

Вы можете сделать блок `catch` условным: если вы предоставите фильтр исключений для своего блока `catch`, он будет перехватывать исключения только тогда, когда условие фильтра выполнено. Листинг 8.6 показывает ситуацию, когда это может оказаться полезно. Он использует клиентский API для Azure Table Storage, службы хранения без SQL, части облачной вычислительной платформы Microsoft Azure. Класс `CloudTable` этого API имеет метод `Execute`, который выдает исключение `StorageException`, если что-то идет не так. Проблема в том, что «что-то идет не так» — это очень широкое понятие и оно охватывает не только ошибки подключения и аутентификации. Вы также увидите это исключение в ситуациях, таких как попытка вставить строку, когда другая строка с теми же ключами уже существует. Это не обязательно

ошибка, так как может происходить в рамках обычного использования в некоторых оптимистичных моделях параллельной обработки данных.

Листинг 8.6. Блок catch с фильтром исключений

```
public static bool InsertIfDoesNotExist(MyEntity item, CloudTable table)
{
    try
    {
        table.Execute(TableOperation.Insert(item));
        return true;
    }
    catch (StorageException x)
    when (x.RequestInformation.HttpStatusCode == 409)
    {
        return false;
    }
}
```

Листинг 8.6 ищет конкретный случай сбоя и возвращает `false` вместо продолжения обработки исключения. Это делается с помощью раздела `when`, содержащего фильтр, который должен быть выражением типа `bool`. Если метод `Execute` выдает исключение `StorageException`, которое не соответствует условию фильтра, исключение будет передаваться как обычно — так, как если бы блока `catch` не было.

Фильтр исключений должен быть выражением, которое производит `bool`. При необходимости он может вызывать внешние методы. Листинг 8.6 просто выбирает пару свойств и выполняет сравнение, но в качестве части выражения можно вызывать любой метод. Тем не менее следует проявлять осторожность и не делать в фильтре ничего, что могло бы вызвать другое исключение². Если это произойдет, оно будет потерянно.

Вложенные блоки try

В случае возникновения исключения в блоке `try`, который не содержит подходящего обработчика, CLR будет продолжать поиск. При необходимости он поднимется вверх по стеку, но вы можете написать несколько наборов обработчиков в одном методе, вложив один `try/catch` в другой блок `try`, как показано в листинге 8.7. `ShowFirstLineLength` вкладывает пару `try/catch`

² Фильтры исключений не могут использовать ключевое слово `await`, которое обсуждается в главе 17.

в блок `try` другой пары `try/catch`. Вложение также может выполняться между методами — метод `Main` будет перехватывать любое исключение `NullReferenceException`, возникающее в методе `ShowFirstLineLength` (оно будет выброшено, если файл полностью пуст — в этом случае вызов `ReadLine` вернет значение `null`).

Листинг 8.7. Вложенная обработка исключений

```
static void Main(string[] args)
{
    try
    {
        ShowFirstLineLength(@"C:\Temp\File.txt");
    }
    catch (NullReferenceException)
    {
        Console.WriteLine("NullReferenceException");
    }
}

static void ShowFirstLineLength(string fileName)
{
    try
    {
        using (var r = new StreamReader(fileName))
        {
            try
            {
                Console.WriteLine(r.ReadLine().Length);
            }

            catch (IOException x)
            {
                Console.WriteLine("Error while reading file: {0}",
                    x.Message);
            }
        }
    }
    catch (FileNotFoundException x)
    {
        Console.WriteLine("Couldn't find the file '{0}'", x.FileName);
    }
}
```

В данном случае я вложил обработчик `IOException`, чтобы он применялся к одной конкретной части задачи: он обрабатывает только ошибки, возни-

кающие при чтении файла после его успешного открытия. Иногда может быть полезно отреагировать на этот сценарий иначе, чем на ошибку, которая помешала вам открыть файл.

Кросс-метод обработки здесь несколько надуманный. `NullReferenceException` можно было бы избежать, проверяя возвращаемое значение `ReadLine` на `null`. Однако лежащий в основе механизм CLR, который этот пример иллюстрирует, чрезвычайно важен. Конкретный блок `try` может определять блоки `catch` только для тех исключений, которые знает, как обрабатывать, что позволяет другим уходить на более высокие уровни.

Позволять исключениям двигаться вверх по стеку часто оказывается правильным. Если ваш метод не может сделать что-то полезное в ответ на обнаружение ошибки, он должен сообщить о проблеме вызывающей стороне, поэтому, если вы не хотите обернуть исключение в другой тип исключения, вы можете пропустить и его.



Если вы знакомы с Java, вам может быть интересно, есть ли в C# что-либо, эквивалентное проверяемым исключениям. Нет. Методы формально не объявляют создаваемые ими исключения, поэтому компилятор не может сообщить вам о том, что вы их не обработали или не объявили, что ваш метод может, в свою очередь, их выбросить.

Вы также можете вложить блок `try` в блок `catch`. Это важно, если есть вероятность того, что ваш обработчик ошибок сам сработает с ошибкой. Например, если ваш обработчик исключений записывает информацию о сбое на диск, это само по себе может вызвать сбой, если имеется проблема с диском.

Некоторые блоки `try` никогда ничего не отлавливают. Запрещено писать блок `try`, за которым ничего не следует, но это что-то не обязательно должно быть блоком `catch`: это может оказаться и блок `finally`.

Блоки `finally`

Блок `finally` содержит код, который выполняется всегда после завершения связанного с ним блока `try`. Он запускается независимо от того, как выполнение покинуло блок `try` — достигнув конца, возвратившись на полпути или выдав исключение. Блок `finally` будет выполняться, даже если

вы используете оператор `goto`, чтобы совершить переход прямо из блока. В листинге 8.8 демонстрируется использование блока `finally`.

Листинг 8.8. Блок `finally`

```
using Microsoft.Office.Interop.PowerPoint;

...

[STAThread]
static void Main(string[] args)
{
    var pptApp = new Application();
    Presentation pres = pptApp.Presentations.Open(args[0]);
    try
    {
        ProcessSlides(pres);
    }
    finally
    {
        pres.Close();
    }
}
```

Это выдержка из утилиты, которую я написал для обработки содержимого файла Microsoft Office PowerPoint. Она просто демонстрирует наиболее внешний код; я опустил сам подробный код обработки, потому что он здесь не нужен (хотя, если вам интересно, полная версия в загружаемых примерах для этой книги экспортирует анимированные слайды в виде видеоклипов). Я показываю его лишь потому, что здесь используется `finally`. Для управления приложением PowerPoint этот пример использует межпрограммное взаимодействие COM. Пример закрывает файл после своего завершения, и причина, по которой я поместил этот код в блок `finally`, заключается в том, что я не хочу, чтобы программа оставляла что-то открытым, если что-то пойдет не так в процессе выполнения. Это важно из-за того, как работает автоматизация COM. Это не похоже на открытие файла, когда ОС после завершения процесса автоматически все закрывает. Если эта программа неожиданно завершит работу, PowerPoint не станет закрывать все, что было открыто, — он просто предположит, что вы намеренно хотели оставить это открытым. (Вы можете сделать это сознательно при создании нового документа, который пользователь затем редактирует.) Я этого не хочу, и закрытие файла в блоке `finally` является надежным способом этого избежать.

Обычно для такого рода вещей вы используете оператор `using`, но API автоматизации PowerPoint на основе COM не поддерживает интерфейс .NET `IDisposable`. Фактически, как мы видели в предыдущей главе, под капотом оператор `using`, как и `foreach`, работает в терминах блоков `finally`. Поэтому вы полагаетесь на механизм `finally` системы обработки исключений, даже когда используете оператор `using` и циклы `foreach`.



Блоки `finally` работают правильно, когда блоки исключений являются вложенными. Если какой-то метод выдает исключение, которое обрабатывается методом, скажем, на пять уровней выше него в стеке вызовов, и если некоторые из промежуточных методов были внутри операторов `using`, циклов `foreach` или блоков `try`, связанных с блоками `finally`, все эти промежуточные блоки `finally` (явно написанные или неявно сгенерированные компилятором) будут выполняться до запуска обработчика.

Конечно, обработка исключений — это только половина дела. Ваш код может сам обнаруживать проблемы, и исключения станут подходящим способом о них сообщить.

Выдача исключений

Выдать исключение очень просто. Вы просто создаете объект исключения соответствующего типа, а затем используете ключевое слово `throw`. Код из листинга 8.9 делает это, когда получает пустой аргумент.

Листинг 8.9. Выдача исключения

```
public static int CountCommas(string text)
{
    if (text == null)
    {
        throw new ArgumentNullException(nameof(text));
    }
    return text.Count(ch => ch == ',');
}
```

CLR делает всю работу за нас. Она собирает информацию, необходимую для исключения, чтобы оно имело возможность сообщать о своем местоположении с помощью таких свойств, как `StackTrace` и `TargetSite`. (Она не рассчитывает их окончательные значения, потому что это довольно затратно.

Она просто проверяет, есть ли у нее информация, необходимая для их выдачи, если это потребуется.) Затем ищет подходящий блок `try/catch` и если какие-либо блоки в конечном итоге должны быть запущены, выполнит их.

Листинг 8.9 иллюстрирует распространенную технику, которую используют при создании исключений, сообщающих о проблеме с аргументом метода. Исключения, такие как `ArgumentNullException`, `ArgumentOutOfRangeException` и их базовый класс `ArgumentException`, могут сообщать имя ошибочного аргумента.

(Это необязательно, потому что иногда нужно сообщать о несоответствии по нескольким аргументам, и в таком случае не нужно указывать ни одного аргумента.) Хорошей идеей будет использовать оператор C# `nameof`. Вы можете использовать его с любым выражением, которое ссылается на именованный элемент, такой как аргумент, переменная, свойство или метод. Он компилируется в строку, содержащую имя элемента.



Многие типы исключений обеспечивают перегрузку конструктора, которая позволяет устанавливать текст `Message`. Более специализированное сообщение может упростить диагностику проблемы, но есть одна вещь, о которой следует позаботиться. Сообщения об исключениях часто попадают в диагностические журналы, а также могут автоматически отправляться системами мониторинга в электронных письмах. Поэтому следует быть осторожным с тем, какую информацию вы помещаете в эти сообщения. Это особенно важно, если ваше программное обеспечение будет использоваться в странах, где действуют законы о защите данных. Размещение в сообщении об исключении информации, которая каким-либо образом относится к конкретному пользователю, может иногда противоречить этим законам.

Я мог бы просто использовать вместо него строковый литерал `text`, но преимущества `nameof` заключаются в том, что он позволяет избежать глупых ошибок (если я наберу `txt` вместо `text`, компилятор скажет мне, что такого символа нет) и помогает избежать проблем, возникающих при переименовании символа. Если я переименую текстовый аргумент в листинге 8.9, я могу легко забыть изменить проверочный строковый литерал. Но используя `nameof` (`text`), я получу ошибку, если изменю имя аргумента, скажем, на `input`, не изменив также `nameof(text)`. Компилятор сообщит, что идентификатора с именем `text` не существует. Если я попрошу Visual Studio переименовать

аргумент, он автоматически обновит все места в коде, которые используют символ, изменив за меня аргумент конструктора исключения на `nameof` (`input`).

Проброс исключений

Иногда полезно написать блок `catch`, который что-то делает в ответ на ошибку, но разрешает ошибке двигаться дальше после завершения этой работы. Очевидный, но неправильный способ проделать это показан в листинге 8.10.

Листинг 8.10. Как не нужно прорабатывать исключение

```
try
{
    DoSomething();
}
catch (IOException x)
{
    LogIOError(x);
    // Следующая строка ПЛОХАЯ!
    throw x; // Не надо так делать!
}
```

Он скомпилируется без ошибок и даже будет работать, но в нем кроется серьезная проблема: он теряет контекст, в котором исключение было сгенерировано первоначально. CLR рассматривает это как совершенно новое исключение (даже если вы повторно используете объект исключения) и сбрасывает информацию о местоположении: `StackTrace` и `TargetSite` сообщают, что ошибка возникла внутри вашего блока `catch`. Это может затруднить диагностику проблемы, потому что вы не сможете увидеть, где она первоначально возникла. Листинг 8.11 показывает, как можно избежать этой проблемы.

Единственная разница между этим примером и листингом 8.10 (кроме удаления предупреждающих комментариев) заключается в том, что я использую ключевое слово `throw` без указания, какой объект использовать в качестве исключения. Вам разрешено делать это только внутри блока `catch`, и тогда он прорабатывает любое исключение, которое обрабатывает блок `catch`. Это означает, что свойства `Exception`, которые сообщают о местоположении исключения, будут по-прежнему ссылаться на исходное местоположение, а не на повторный проброс.

Листинг 8.11. Проброс без потери контекста

```
try
{
    DoSomething();
}
catch (IOException x)
{
    LogIOError(x);
    throw;
}
```



В .NET Framework (т. е. если вы не используете .NET Core) листинг 8.11 решает проблему не полностью. Хотя точка, в которой было сгенерировано исключение (что в данном примере происходит где-то внутри метода `DoSomething()`), будет сохранена, часть трассировки стека, показывающая, до какой точки добрался метод в листинге 8.11, не сохранится. Вместо того чтобы сообщить, что метод достиг строки, которая вызывает `DoSomething`, он укажет, что он был в строке, содержащей `throw`. Немного странный побочный эффект этого состоит в том, что трассировка стека заставит все выглядеть так, как будто метод `DoSomething` был вызван ключевым словом `throw`. В .NET Core этой проблемы нет.

Существует еще одна проблема, связанная с контекстом, о которой следует помнить при обработке исключений, которые вы захотите пробросить. Она связана с тем, как CLR передает информацию в *Windows Error Reporting* (WER), компонент, который вступает в действие при сбое приложения в Windows³. В зависимости от того, как настроена ваша машина, WER может отображать диалоговое окно сбоя, включающее варианты перезапуска приложения, сообщения о сбое в Microsoft, отладки приложения или просто его завершения. В дополнение ко всему этому при сбое приложения Windows WER сохраняет несколько фрагментов информации для определения места сбоя. Для приложений .NET это включает в себя имя, версию и временную метку компонента, в котором произошел сбой, а также тип исключения. Кроме того, он определяет не только метод, но и его смещение в промежуточном языке, на котором было сгенерировано исключение. Эти фрагменты информации иногда называют контейнерными значениями. Если приложение дважды аварийно завершает работу с одними и теми же значениями, эти два сбоя

³ Некоторые называют WER именем более старого механизма отчетов о сбоях Windows: Dr. Watson.

попадают в один и тот же контейнер, что означает, что они в некотором смысле рассматриваются как один и тот же сбой.

Значения аварийного контейнера не отображаются как открытые свойства исключений, но их можно увидеть в журнале событий Windows для любого исключения, которое достигло обработчика CLR по умолчанию. В приложении *Windows Event Viewer* эти записи журнала отображаются в разделе **Application** в **Windows Logs**. Столбцы **Source** и **Event ID** для этих записей будут содержать **WER** и **1001** соответственно. **WER** сообщает о различных видах сбоев, поэтому, если вы откроете запись в журнале **WER**, она будет содержать значение **Event Name**. Для сбоев .NET это будет **CLR20r3**. Название и версию сборки достаточно легко определить, как и тип исключения. Поиск метода не так прост: он находится в строке, помеченной как **P7**, но это просто число, основанное на маркере метаданных метода. Для выяснения того, к какому методу оно относится, в инструменте **ILDASM**, поставляемом с *Visual Studio*, есть параметр командной строки, предназначенный для сообщения маркеров метаданных для всех ваших методов.



Способ получения доступа к накопленным данным о сбоях зависит от типа приложения, которое вы пишете. Для бизнес-приложения, которое работает только внутри вашего предприятия, вы, вероятно, захотите запустить собственный сервер отчетов об ошибках, но, если приложение работает вне вашего административного контроля, вы можете использовать серверы сбоев, принадлежащие Microsoft. Необходимо пройти процесс на основе сертификатов для проверки того, что вы имеете право на эти данные, но, как только вы преодолели это препятствие, Microsoft с готовностью покажет вам все зарегистрированные сбои для ваших приложений, отсортированные по размеру контейнера.

Получение этой информации из журнала событий Windows очень хорошо подходит для кода, выполняемого на компьютерах, которые вы контролируете. (Или вы можете предпочесть использовать более прямые способы мониторинга таких приложений с использованием для сбора телеметрии таких систем, как *Microsoft Application Insights*, и в этом случае **WER** не так полезен.) Роль **WER** становится более заметной в приложениях, которые могут работать на компьютерах за пределами вашего контроля, например в локальных приложениях с пользовательским интерфейсом или в консольных приложениях. Компьютеры могут быть настроены на загрузку отчетов о сбоях в службу отчетов об ошибках, и, как правило, туда отправляются

только контейнерные значения, хотя службы могут запрашивать и другие данные, если конечный пользователь не против. Анализ контейнеров может быть полезен при принятии решения о приоритете исправления ошибок: имеет смысл начать с самого большого контейнера, потому что именно с этим сбоем пользователи сталкиваются чаще всего. (Или, по крайней мере, с которыми сталкиваются пользователи, не отключившие отчеты о сбоях.) Я всегда включаю их на своих компьютерах, потому что хочу, чтобы ошибки, с которыми сталкиваюсь в используемых программах именно я, были исправлены первыми.)

Определенная тактика обработки исключений может заставить систему аварийных контейнеров работать неправильно. Если вы пишете обычный код обработки ошибок, который связывается со всеми исключениями, существует риск того, что WER решит, что ваше приложение способно аварийно завершаться только в этом общем обработчике, что будет означать, что все виды сбоев попадут в один контейнер. Этого можно избежать, но, чтобы это сделать, вам нужно понять, как ваш код обработки исключений влияет на данные контейнера аварийных сбоев WER.

Если исключение поднимается до вершины стека без обработки, WER получит точную картину того, где именно произошел сбой. Но все может разладиться, если вы отловите исключение, прежде чем в конечном итоге разрешить ему (или другому исключению) подниматься дальше по стеку. Немного удивляет, что .NET успешно сохранит его расположение для WER, даже если вы используете неправильный подход, показанный в листинге 8.10. (Лишь с точки зрения .NET внутри этого приложения теряется контекст исключения — `StackTrace` покажет местоположение повторного проброса. Таким образом, WER не обязательно сообщает о том же месте сбоя, которое код .NET увидит в объекте исключения.) Это та же история, когда вы оборачиваете новое исключение в `InnerException`: .NET будет использовать расположение этого внутреннего исключения для значений контейнера сбоев.

Это означает, что сохранить контейнер WER относительно легко. Единственный способ потерять исходный контекст — это либо полностью обработать исключение (т. е. не аварийно завершить работу), либо написать блок `catch`, который обрабатывает исключение, а затем создает новое, не передавая исходное в качестве `InnerException`.

Хотя листинг 8.11 сохраняет исходный контекст, этот подход имеет ограничение: вы можете прорабатывать исключение только изнутри блока, в котором его перехватили. Поскольку асинхронное программирование становится все

более распространенным, все чаще встречаются исключения, возникающие в случайном рабочем потоке. Нам нужен надежный способ для захвата полного контекста исключения и для возможности пробросить его вместе с этим полным контекстом через произвольное количество времени и, возможно, из другого потока.

Класс `ExceptionDispatchInfo` решает все эти проблемы. Если вы из блока `catch` вызываете его статический метод `Capture`, передавая ему текущее исключение, он захватывает полный текст, включая информацию, требуемую `WER`. Метод `Capture` возвращает экземпляр `ExceptionDispatchInfo`. Когда вы будете готовы пробросить исключение, вы можете вызвать метод `Throw` этого объекта, и `CLR` пробросит исключение с полным исходным контекстом. В отличие от механизма, показанного в листинге 8.11, вам не нужно находиться внутри блока `catch` при повторном пробросе. Вам даже не нужно находиться в потоке, из которого изначально было вызвано исключение.



Если вы используете ключевые слова `async` и `await`, описанные в главе 17, то они используют `ExceptionDispatchInfo` для гарантии, что контекст исключения сохраняется правильно.

Быстрый вылет

Некоторые ситуации требуют решительных действий. Если вы обнаружили, что ваше приложение находится в безнадежно поврежденном состоянии, создать исключение может оказаться недостаточно, поскольку всегда есть вероятность, что что-то его обработает, а затем попытаться продолжить работу. Это может привести к повреждению персистентного состояния — возможное недопустимое состояние в памяти приведет к тому, что ваша программа запишет неверные данные в базу данных. Возможно, лучше сразу же катапультироваться, не рискуя нанести какой-то более ощутимый ущерб.

Класс `Environment` предоставляет метод `FailFast`. Если вы его вызовете, `CLR` прекратит работу вашего приложения. (Если вы работаете в Windows, то ОС также запишет сообщение в журнал событий Windows и предоставит сведения для `WER`.) Вы можете передать строку для включения в запись журнала событий, а также передать исключение. Windows в таком случае также занесет сведения об исключении в журнал, включая значения контейнера `WER` для точки, в которой было сгенерировано исключение.

Типы исключений

Когда ваш код обнаруживает проблему и выдает исключение, нужно выбрать, какой тип исключения подходит. Вы можете определить свои собственные типы исключений, но библиотека классов .NET определяет большое количество типов исключений, поэтому во многих ситуациях вы можете просто выбрать существующий. Существуют сотни типов исключений, поэтому полный список здесь помещать нецелесообразно; увидеть полный набор можно в онлайн-документации для класса `Exception`, где перечислены производные типы. Однако есть определенные типы, о которых важно знать.

Библиотека классов определяет класс `ArgumentException`, который является базовым для нескольких исключений, указывающих, что метод был вызван с неверными аргументами. В листинге 8.9 использовалось исключение `ArgumentNullException`, а кроме него имеется еще `ArgumentOutOfRangeException`. Базовое `ArgumentException` определяет свойство `ParamName`, содержащее имя параметра, в который был передан неверный аргумент. Это важно для методов с несколькими аргументами, потому что вызывающая сторона должна знать, какой из них был неправильным. Все эти типы исключений имеют конструкторы, которые позволяют указать имя параметра, и один из них вы можете увидеть в листинге 8.9. Базовое `ArgumentException` является реальным классом, поэтому, если аргумент неверен, но этот случай не охватывается ни одним из производных типов, можно просто вызвать базовое исключение, предоставив текстовое описание проблемы.

Помимо только что описанных типов общего назначения некоторые API определяют более специализированные производные исключения для аргументов. Например, пространство имен `System.Globalization` содержит тип исключения с именем `CultureNotFoundException`, который является производным от `ArgumentException`. Вы можете сами сделать что-то подобное, и существует две причины, по которым вам это может понадобиться. Если вы можете предоставить дополнительную информацию о том, почему аргумент недействителен, вам потребуется пользовательский тип исключения, дающий возможность присоединить эту информацию к исключению. (`CultureNotFoundException` предоставляет три свойства, описывающих аспекты информации о культуре, для которой осуществлялся поиск.) В другом случае может оказаться, что вызывающая сторона может особым образом обрабатывать определенную форму ошибки аргумента. Зачастую исключение

аргумента просто указывает на ошибку программирования, но в ситуациях, когда оно может указывать на проблему среды или конфигурации (например, отсутствие установленных языковых пакетов), разработчикам может потребоваться решить эту конкретную проблему по-другому. Использование базового `ArgumentException` было бы в этом случае бесполезным, потому что при этом трудно выделить конкретную ошибку, которую нужно обработать, как и любую другую проблему с аргументами.

Некоторым методам может понадобиться выполнить работу, которая способна привести к множественным ошибкам. Возможно, вы выполняете какое-то пакетное задание, и если некоторые отдельные задачи пакета вызывают сбои, вы бы хотели прервать их, но дать остальным возможность выполниться, сообщив при этом обо всех сбоях в конце. Для таких сценариев подойдет `AggregateException`. Он расширяет идею `InnerException` базового `Exception`, добавляя свойство `InnerExceptions`, которое возвращает коллекцию исключений.



Если вы будете выполнять вложенную работу, которая может создать `AggregateException` (например, запуская пакет внутри пакета), некоторые из ваших внутренних исключений могут тоже получить тип `AggregateException`. Это исключение предлагает метод `Flatten`, который рекурсивно просматривает любые такие вложенные исключения и создает один плоский список со всеми вложенными объектами, возвращая исключение `AggregateException` с этим списком в качестве своего `InnerException`.

Другой часто используемый тип — `InvalidOperationException`. Его обычно выбрасывают, если кто-то пытается проделать с объектом то, что он не поддерживает в своем текущем состоянии. Например, предположим, что вы написали класс, представляющий запрос, который можно отправить на сервер. Вы можете спроектировать его таким образом, чтобы каждый экземпляр можно было использовать только единожды. Поэтому, если запрос уже был отправлен, попытка последующего изменения запроса будет ошибкой, и к этому случаю как раз подойдет указанное исключение. Еще один важный пример — если ваш тип реализует `IDisposable`, а кто-то пытается использовать экземпляр после его удаления. Это настолько распространенный случай, что существует специальный тип, производный от `InvalidOperationException`, называемый `ObjectDisposedException`.

Следует упомянуть о различии между `NotImplementedException` и похоже выглядящим, но семантически другим, `NotSupportedException`. Последнее должно вызываться, когда этого требует интерфейс. Например, интерфейс `IList<T>` определяет методы для изменения коллекций, но не требует, чтобы коллекции были модифицируемыми. Вместо этого он предписывает членам коллекций, доступных только для чтения, выдавать исключение `NotSupportedException` при попытке их изменения. Реализация `IList<T>` может выдавать его и все еще считаться завершенной, тогда как `NotImplementedException` означает, что чего-то не хватает. Чаще всего вы увидите это в коде, сгенерированном Visual Studio. Среда IDE способна создавать методы-заглушки, когда вы попросите ее сгенерировать реализацию интерфейса или предоставить обработчик событий. Она генерирует этот код, чтобы избавить вас от необходимости писать полное объявление метода, но реализация тела метода — это ваша задача. Поэтому Visual Studio зачастую создает методы, которые выдают это исключение, чтобы вы случайно не оставили их пустыми.

Обычно вы удаляете весь код, который выдает `NotImplementedException` перед отправкой заказчику, заменив его соответствующими реализациями. Тем не менее есть ситуация, в которой вы все же можете выдать это исключение. Предположим, что вы написали библиотеку, содержащую абстрактный базовый класс, а ваши клиенты пишут основанные на нем классы. Когда вы выпускаете новые версии библиотеки, вы можете добавлять в этот базовый класс новые методы. Теперь представьте, что вы хотите добавить новую библиотечную функцию, для которой, как вам кажется, имеет смысл добавить новый абстрактный метод в ваш базовый класс. Это серьезный шаг, ведь существующий код, который успешно наследовался от старой версии класса, больше не будет работать. Вы можете избежать этой проблемы, предоставив виртуальный метод вместо абстрактного, но что, если вы не можете предоставить никакой полезной реализации по умолчанию? Именно в этом случае вы можете написать базовую реализацию, которая генерирует исключение `NotImplementedException`. Код, созданный для старой версии библиотеки, не будет пытаться использовать новую функцию, поэтому никогда не будет пытаться вызвать этот метод. Но если клиент попытается использовать новую библиотечную функцию без переопределения соответствующего метода в своем классе, он увидит это исключение. Другими словами, это способ потребовать реализации: вы переопределяете этот метод, если и только тогда, когда хотите использовать функцию, которую он представляет. (Вы можете использовать тот же подход при добавлении новых

членов в интерфейс, если вы используете недавно добавленную в C# 8.0 поддержку реализаций интерфейса по умолчанию.)

Конечно, в структуре есть и другие, более специализированные исключения, и вы всегда должны пытаться использовать такое, которое соответствует проблеме, о которой вы хотите сообщить. Однако иногда вам потребуется сообщить об ошибке, для которой в библиотеке классов .NET нет подходящего исключения. В этом случае вам придется писать собственный класс исключений.

Пользовательские исключения

Минимальное требование для пользовательского типа исключения состоит в том, что он должен быть производным от `Exception` (прямо или косвенно). Тем не менее существуют определенные рекомендации по дизайну. Первое, на что нужно обратить внимание, — это непосредственный базовый класс: если вы посмотрите на встроенные типы исключений, вы заметите, что многие из них наследуются от `Exception` лишь косвенно, либо через `ApplicationException`, либо через `SystemException`. Обоих следует избегать. Изначально они появились с целью разграничить исключения, создаваемые приложениями и .NET. Однако время показало, что это не самое удачное разделение. При различных сценариях некоторые исключения могут быть вызваны как приложениями, так и .Net. В любом случае в обычных обстоятельствах не стоит писать обработчик, который перехватывает все исключения приложения, но только часть системных или наоборот. В настоящее время рекомендации по проектированию библиотеки классов советуют вам избегать этих двух базовых типов.

Пользовательские классы исключений обычно наследуются непосредственно от `Exception`, если только они не представляют специализированную форму уже существующего исключения. Например, мы уже видели, что `ObjectDisposedException` — это частный случай `InvalidOperationException` и библиотека классов определяет несколько более специализированных производных этого базового класса, таких как `ProtocolViolationException` для сетевого кода.

Если проблема, о которой вы хотите сообщить из вашего кода, явно является примером какого-то существующего типа исключения, который следует сделать более специализированным, то вам стоит наследовать этот существующий тип.

Хотя базовый класс `Exception` имеет конструктор без параметров, его использовать не рекомендуется. Исключения должны содержать наглядное текстовое описание ошибки, поэтому все конструкторы ваших пользовательских исключений должны вызывать один из конструкторов `Exception`, которые принимают строку. Вы можете либо жестко закодировать строку сообщения в вашем производном классе, либо определить конструктор, который принимает сообщение, передавая его базовому классу. Типы исключений обычно содержат оба варианта, хотя это может быть пустой тратой усилий, если ваш код использует только один из конструкторов⁴. Это зависит от того, может ли другой код вызвать ваше исключение или это доступно только вашему.

Также стоит добавить конструктор, принимающий другое исключение, которое станет значением свойства `InnerException`. Опять же, если вы пишете исключение только для себя, добавлять этот конструктор не нужно, пока он вам на самом деле не понадобится, но если ваше исключение — часть библиотеки многократного использования, это частая функция. Листинг 8.12 показывает гипотетический пример, который содержит различные конструкторы, а также тип перечисления, используемый свойством, которое добавляет исключение.

Листинг 8.12. Пользовательское исключение

```
public class DeviceNotReadyException : InvalidOperationException
{
    public DeviceNotReadyException(DeviceStatus status)
        : this("Device status must be Ready", status)
    {
    }

    public DeviceNotReadyException(string message, DeviceStatus status)
        : base(message)
    {
        Status = status;
    }

    public DeviceNotReadyException(string message, DeviceStatus status,
```

⁴ Вместо того чтобы жестко задавать ее в коде, можно попробовать найти локализованную строку с помощью средств в пространстве имен `System.Resources`. Это делают все исключения в библиотеке классов .NET. Это не обязательно, потому что не все программы работают сразу в нескольких регионах, но даже в случае тех, которые это делают, сообщения об исключениях не всегда будут показываться конечным пользователям.

```
        Exception innerException)
    : base(message, innerException)
{
    Status = status;
}

public DeviceStatus Status { get; }

public enum DeviceStatus
{
    Disconnected,
    Initializing,
    Failed,
    Ready
}
```

Основанием для создания пользовательского исключения здесь является то, что эта конкретная ошибка может сообщить нам что-то еще кроме факта, что нечто не было в нужном состоянии. Исключение предоставляет информацию о состоянии объекта в момент, когда операция не удалась.

Раньше руководство по проектированию .NET Framework рекомендовало, чтобы исключения были сериализуемыми. Исторически это должно было позволить им переходить между доменами приложений. Домен приложения — это изолированный контекст выполнения; однако на сегодня они устарели, так как не поддерживаются .NET Core. Тем не менее еще существуют типы приложений, в которых сериализация исключений полезна, особенно архитектуры на основе микросервисов, например на основе Akka.NET (см. <https://github.com/akkadotnet/akka.net>) или Microsoft.ServiceFabric, в которой одно приложение выполняется в нескольких процессах, часто распределяясь по множеству разных компьютеров. Делая исключение сериализуемым, вы позволяете ему пересекать границы процесса. В этом случае нельзя использовать непосредственно исходный объект исключения, но сериализация позволяет встроить копию исключения в целевой процесс.

Таким образом, хотя сериализация больше и не рекомендуется для всех типов исключений, она остается полезной в случае исключений, которые могут использоваться в подобных многопроцессорных средах. По этой причине большинство типов исключений в .NET Core продолжают поддерживать сериализацию. Если вам этого не требуется, вы можете не делать свои исключения сериализуемыми, но, так как это довольно распространено,

я опишу изменения, которые вам нужно будет сделать. Во-первых, вам нужно добавить атрибут `[Serializable]` перед объявлением класса. Затем вам нужно переопределить метод, определяемый `Exception`, который управляет сериализацией. Наконец, вы должны написать специальный конструктор, который будет использоваться при десериализации вашего типа. В листинге 8.13 показано, какие члены необходимо добавить, чтобы пользовательское исключение в листинге 8.12 поддерживало сериализацию. Метод `GetObjectData` просто сохраняет текущее значение свойства `Status` в контейнере имя/значение, предоставляемом во время сериализации. Он получает это значение в конструкторе, который вызывается во время десериализации.

Листинг 8.13. Добавление поддержки сериализации

```
public override void GetObjectData(SerializationInfo info,
                                    StreamingContext context)
{
    base.GetObjectData(info, context);
    info.AddValue("Status", Status);
}

protected DeviceNotReadyException(SerializationInfo info,
                                  StreamingContext context)
    : base(info, context)
{
    Status = (DeviceStatus) info.GetValue("Status", typeof(DeviceStatus));
}
```

Необработанные исключения

Ранее вы видели поведение консольного приложения по умолчанию в том случае, когда ваше приложение выдает исключение, которое оно не обрабатывает. Оно отображает тип исключения, сообщение и трассировку стека, после чего завершает процесс. Это происходит независимо от того, было ли необработанное исключение в основном потоке, или в потоке, который вы создали явно, или даже в потоке пула потоков, который CLR для вас создал.

Имейте в виду, что за прошедшие годы в поведение необработанных исключений было внесено несколько изменений, которые все еще имеют определенную ценность, поскольку при желании вы можете вернуть старое поведение. До .NET 2.0 потоки, созданные для вас CLR, проглатывали

исключения, не сообщая о них и не прерывая работу. Иногда вы можете столкнуться со старыми приложениями, которые все еще работают подобным образом: если у приложения есть файл конфигурации в стиле .NET Framework (в виде XML, подобно использованному в главе 7 для настройки GC), который содержит элемент `legacyUnhandledExceptionPolicy` с атрибутом `enabled="1"`, то возвращается старое поведение .NET v1, означающее, что необработанные исключения могут тихо исчезать. .NET 4.5 сделал шаг в противоположную сторону. Если раньше для параллельных задач вы использовали класс `Task` (описанный в главе 16) вместо того, чтобы напрямую использовать потоки или пул потоков, любые необработанные исключения внутри задач приводили к завершению процесса, но в .NET 4.5 это больше не является поведением по умолчанию. Вы можете вернуться к старому поведению через файл конфигурации. (Подробности см. в главе 16.)

CLR предоставляет способ обнаружить, когда необработанные исключения достигают вершины стека. Класс `AppDomain` содержит событие `UnhandledException`, которое CLR вызывает, когда это происходит в каком-либо потоке. Я буду описывать события в главе 9, но листинг 8.14 забегает вперед и показывает, как обрабатывать это событие⁵. Он также выдает необработанное исключение, чтобы опробовать обработчик.

Листинг 8.14. Уведомления о необработанных исключениях

```
static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += OnUnhandledException;

    // Умышленный сбой для демонстрации UnhandledException
    throw new InvalidOperationException();
}

private static void OnUnhandledException(object sender,
    UnhandledExceptionEventArgs e)
{
    Console.WriteLine($"An exception went unhandled: {e.ExceptionObject}");
}
```

⁵ Хотя .NET Core не поддерживает создание новых доменов приложений, он все же предоставляет класс `AppDomain`, поскольку содержит важный функционал, такой как это событие. Он предоставляет единственный экземпляр через `AppDomain.CurrentDomain`.

Когда обработчик получает уведомление, исключение останавливать уже поздно — CLR завершит процесс вскоре после вызова вашего обработчика. Основная причина, по которой существует это событие, заключается в предоставлении места для размещения кода журналирования, чтобы вы в диагностических целях могли записать определенную информацию о сбое. В принципе, вы можете также попытаться сохранить любые несохраненные данные, чтобы облегчить восстановление, если программа перезапустится, но действовать следует осторожно: если вызван ваш обработчик необработанных исключений, то ваша программа по определению находится в подозрительном состоянии, поэтому любые сохраняемые данные могут оказаться некорректными.

Некоторые среды разработки приложений предоставляют свои собственные способы обработки необработанных исключений. Например, это делают каркасы разработки пользовательского интерфейса (например, *Windows Forms* или *WPF*) для настольных приложений Windows, отчасти потому, что стандартное поведение записи подробностей в консоль не очень полезно в случае приложений, которые не отображают окно консоли. Этим приложениям необходимо запускать цикл обработки сообщений, чтобы отвечать на ввод пользователя и системные сообщения. Он проверяет каждое сообщение и может вызвать один или несколько методов в вашем коде. В этом случае он упаковывает каждый вызов в блок `try`, чтобы иметь возможность перехватывать любые исключения, которые способен выдать ваш код. Библиотеки классов могут вместо этого отображать информацию об ошибках в окне. А веб-фреймворкам, таким как *ASP.NET Core*, требуется другой механизм: как минимум они должны генерировать ответ, который указывает на ошибку на стороне сервера способом, рекомендуемым спецификацией HTTP.

Это означает, что событие `UnhandledException`, используемое в листинге 8.14, может не возникать, когда необработанное исключение покидает ваш код, поскольку может быть перехвачено платформой. Если вы используете среду разработки приложений, вам следует проверить, предоставляет ли она собственный механизм для работы с необработанными исключениями.

Например, приложения *ASP.NET Core* могут делать обратный вызов метода с именем `UseExceptionHandler`. *WPF* имеет свой собственный класс `Application`, и именно его событие `DispatcherUnhandledException` следует использовать. Аналогично *Windows Forms* предоставляет класс `Application` с элементом `ThreadException`.

Даже когда вы используете эти библиотеки классов, их механизмы необработанных исключений имеют дело только с исключениями, возникающими в подконтрольных потоках. Если вы создадите новый поток и сгенерируете необработанное исключение, оно будет отображаться в событии `UnhandledException` класса `AppDomain`, поскольку библиотеки классов не контролируют весь CLR.

Итог

В .NET об ошибках, как правило, сообщают с помощью исключений, за вычетом ряда сценариев, где сбои считаются обычным делом, а затраты на исключения оказываются высокими по сравнению с выполняемой задачей. Исключения позволяют коду обработки ошибок располагаться отдельно от рабочего кода. Они также затрудняют игнорирование ошибок — непредвиденные ошибки распространяются вверх по стеку и в конечном итоге приводят к завершению программы и выдаче отчета об ошибках. Блоки `catch` позволяют нам обрабатывать те исключения, которые мы можем предвидеть. (Вы также можете использовать их для беспорядочного перехвата всех исключений, но это, как правило, плохая идея — если вы не знаете, почему произошло то или иное исключение, вы не можете точно знать, как после него безопасно восстановиться.) Блоки `finally` дают способ безопасного выполнения очистки независимо от того, выполняется код успешно или с исключениями. Библиотека классов .NET определяет множество полезных типов исключений, но при необходимости мы можем писать и свои собственные.

До сих пор в главах мы рассмотрели основные элементы кода, классов и других пользовательских типов, коллекций и обработки ошибок. Есть еще одна особенность системы типов C#, на которую стоит обратить внимание: особый вид объекта, называемый делегатом.

ГЛАВА 9

Делегаты, лямбды и события

Наиболее распространенный способ использования API — это вызов методов и свойств, предоставляемых его классами. Однако иногда все должно проходить в обратном порядке — API может потребоваться вызвать ваш код. В главе 5 я показывал функции поиска, предлагаемые массивами и списками. Чтобы использовать их, я написал метод, который возвращал значение `true`, когда его аргумент соответствовал моим критериям и соответствующие API вызывали мой метод для каждого проверяемого элемента. Не все обратные вызовы являются срочными. Асинхронные API могут вызывать метод в нашем коде после завершения длительной работы. В клиентском приложении мне потребуется, чтобы мой код запускался, когда пользователь определенным образом взаимодействует с определенными визуальными элементами, например нажимает кнопку.

Интерфейсы и виртуальные методы могут задействовать обратные вызовы. В главе 4 я показал интерфейс `IComparer<T>`, который определяет единственный метод, `CompareTo`. Он вызывается такими методами, как `Array.Sort`, когда мы хотим задать свой порядок сортировки. Вполне можно представить себе библиотеку пользовательского интерфейса, которая определяла бы интерфейс `IClickHandler`, содержащий метод `Click` и, возможно, также `DoubleClick`. Библиотека может потребовать от нас реализации этого интерфейса, если мы хотим получать уведомления о нажатиях кнопок.

На самом деле ни одна из платформ .NET UI не использует подход, основанный на интерфейсе, потому что он становится громоздким, когда требуется несколько видов обратной связи. Одиночный и двойной щелчок — это вершина айсберга в случае взаимодействия с пользователем, так как в приложениях WPF каждый элемент пользовательского интерфейса может поддерживать более 100 видов уведомлений. В большинстве случаев вам нужно обрабатывать только одно или два события от любого конкретного элемента, поэтому интерфейс с 100 реализуемыми методами будет крайне неудобен.

Распределение уведомлений по нескольким интерфейсам может немного сгладить это неудобство. Кроме того, могла бы оказаться полезной под-

держка в C# 8.0 реализации интерфейса по умолчанию, которая позволила бы обеспечить пустые реализации по умолчанию для всех обратных вызовов и дать возможность переопределять только те из них, которые нас интересуют. (Если вам нужно ориентироваться на среды выполнения более ранние, чем .NET Core 3.0, в котором впервые появилась эта функция, вы можете вместо этого добавить базовый класс с виртуальными методами.) Но даже с этими улучшениями у этого объектно ориентированного подхода есть серьезный недостаток. Представьте себе пользовательский интерфейс с четырьмя кнопками. В гипотетической библиотеке пользовательского интерфейса, в которой бы использовался только что описанный подход, для обработки `Click` каждой кнопки вам потребовались бы четыре различные реализации интерфейса `IClickHandler`. Один класс может реализовать любой конкретный интерфейс только единожды, поэтому вам пришлось бы написать четыре класса. Это выглядит чрезмерно громоздко, тогда как все, что мы на самом деле хотим сделать, — это сообщить кнопке, чтобы она при нажатии вызывала определенный метод.

C# предоставляет гораздо более простое решение в виде делегата, который является ссылкой на метод. Если вам для чего-то нужно, чтобы библиотека вызывала ваш код, вы просто передаете делегат, ссылающийся на требуемый метод. Я показал пример этого в главе 5 и воспроизвел его в листинге 9.1. Он находит индекс первого ненулевого элемента в массиве `int[]`.

Листинг 9.1. Поиск в массиве с использованием делегата

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins) =>
    Array.FindIndex(bins, IsGreaterThanZero);

private static bool IsGreaterThanZero(int value) => value > 0;
```

На первый взгляд пример выглядит очень простым: для второго параметра `Array.FindIndex` требуется метод, который он может вызвать, чтобы проверить совпадение конкретного элемента, так что в качестве аргумента я передал свой метод `IsGreaterThanZero`. Но что на самом деле означает передача метода и как это согласуется с системой типов CTS в .NET?

Делегаты

В листинге 9.2 показано объявление метода `FindIndex`, использованного в листинге 9.1. Первый параметр — это массив для поиска, но нас интересует второй, тот, через который я передал метод.

Листинг 9.2. Метод с делегатом в качестве параметра

```
public static int FindIndex<T>(  
    T[] array,  
    Predicate<T> match)
```

Тип второго аргумента метода — `Predicate<T>` где `T` — тип элемента массива, и поскольку в листинге 9.1 используется `int[]`, это будет `Predicate<int>`. (Если у вас нет опыта ни в формальной логике, ни в информатике, этот тип использует слово предикат в смысле функции, которая определяет, является ли что-то истинным или ложным. Например, у вас может быть предикат, который сообщает вам, является ли число четным. Предикаты часто используются в подобных операциях отбора.) Листинг 9.3 показывает, как этот тип определяется. Это полное определение, а не выдержка; если вы хотите написать тип, который был бы эквивалентен `Predicate<T>`, то это все, что вам нужно написать.

Листинг 9.3. Тип делегата `Predicate<T>`

```
public delegate bool Predicate<in T>(T obj);
```

Разберем листинг 9.3 по частям. Начинаем мы, как обычно, с определений типов, доступности и можем при этом использовать те же ключевые слова, которые используем для других типов, например `public` или `internal`. (Как и любой тип, типы делегатов могут быть вложены в какой-то другой тип, поэтому вы также можете использовать `private` или `protected`.) Далее следует ключевое слово `delegate`, которое просто сообщает компилятору C#, что мы определяем тип делегата. Оставшаяся часть определения выглядит как объявление метода, и это не случайно. У нас есть возвращаемый тип `bool`. Вы помещаете имя типа делегата туда, где обычно записано имя метода. Угловые скобки указывают, что это обобщенный тип с единственным аргументом типа `T`, а ключевое слово `in` указывает, что `T` является контравариантным. (Глава 6 описывает контравариантность.) Наконец, сигнатуре метода имеет единственный параметр этого типа.

Делегаты — это специальные типы в .NET, и они работают совсем не так, как классы или структуры. Компилятор создает внешне нормально выглядящее определение класса с различными членами, которые мы рассмотрим более подробно позже, но все члены пусты — С # не создает промежуточного языка ни для одного из них. CLR предоставляет их реализацию во время выполнения.

Экземпляры типов делегатов обычно называют просто делегатами, и они ссылаются на методы. Метод совместим с конкретным типом делегата (т. е. может быть вызван его экземпляром), если его сигнатура совпадает с заданной. Метод `IsGreater ThanZero` в листинге 9.1 принимает `int` и возвращает `bool`, поэтому он совместим с `Predicate<int>`. Совпадение не обязательно должно быть точным. Если для типов параметров доступны неявные ссылочные преобразования, вы можете использовать более общий метод. (Это не имеет ничего общего с тем, что `T` контравариантен. Вариантность делает доступными определенные неявные ссылочные преобразования между типами делегатов, созданными из одного и того же несвязанного обобщенного делегата с аргументами разных типов. Здесь мы обсуждаем диапазон сигнатур методов, приемлемых для одного типа делегата.) Например, метод с возвращаемым типом `bool` и единственным параметром типа `object` будет совместим с `Predicate<object>`, но поскольку такой метод может принимать строковые аргументы, он также будет совместим и с `Predicate<string>`. (Но не будет совместим с `Predicate<int>`, потому что не существует неявного преобразования ссылок из `int` в `object`. Существует неявное преобразование, но это упаковка, а не преобразование ссылки.)

Создание делегата

Для создания делегата вы можете использовать ключевое слово `new`. Там, куда вы обычно помещаете аргументы конструктора, вы можете записать имя совместимого метода. Листинг 9.4 создает `Predicate<int>`, поэтому ему нужен метод с типом возврата `bool`, который принимает `int`, и, как мы только что видели, метод `IsGreater ThanZero` в листинге 9.1 отвечает всем требованиям. (Вы можете использовать этот код только там, где `IsGreater ThanZero` находится в области видимости, т. е. внутри того же класса.)

Листинг 9.4. Построение делегата

```
var p = new Predicate<int>(IsGreater ThanZero);
```

На практике мы редко используем `new` для делегатов. Это необходимо только в тех случаях, когда компилятор не может определить тип делегата. Выражения, которые ссылаются на методы, необычны тем, что у них нет присущего типа — выражение `IsGreater ThanZero` совместимо с `Predicate<int>`, но есть и другие совместимые типы делегатов. Вы можете определить свой собственный необобщенный тип делегата, который принимает `int` и возвращает `bool`.

Позже в этой главе я покажу семейство типов делегатов `Func`; вы могли бы сохранить ссылку на `IsGreater ThanZero` в делегате `Func<int, bool>`.

Таким образом, `IsGreater ThanZero` не имеет своего собственного типа, поэтому компилятору необходимо знать, какой именно тип делегата нам нужен. В листинге 9.4 делегат присваивается переменной, объявленной с помощью `var`, которая ничего не сообщает компилятору об используемом типе, и именно поэтому мне пришлось явно указать это с помощью синтаксиса конструктора.

В случаях, когда компилятор знает, какой тип требуется, он способен неявно преобразовать имя метода в целевой тип делегата. В листинге 9.5 объявляется переменная с явным типом, поэтому компилятор знает, что требуется `Predicate<int>`. Он компилируется в тот же код, что и листинг 9.4. Листинг 9.1 опирается на тот же механизм — компилятор знает, что вторым аргументом для `FindIndex` является `Predicate<T>`, поскольку мы предоставляем первый аргумент типа `int[]`, он понимает, что `T` — это `int`, поэтому полным типом второго аргумента является `Predicate<int>`. Разобравшись в этом, он использует те же встроенные правила неявного преобразования для создания делегата, как в листинге 9.5.

Листинг 9.5. Неявное конструирование делегата

```
Predicate<int> p = IsGreater ThanZero;
```

Когда код ссылается на метод по имени, как это происходит в данном случае, имя технически называется правилом, поскольку для одного имени может существовать несколько перегрузок. Компилятор сужает этот диапазон, находя наилучшее возможное совпадение, аналогично тому, как он выбирает перегрузку при вызове метода. Как и в случае вызова метода, совпадений может либо не оказаться, либо будет несколько одинаково хороших совпадений. В обоих этих случаях компилятор выдаст ошибку.

Правила могут принимать несколько форм. В приведенных выше примерах я использовал неполное имя метода, которое работает только тогда, когда рассматриваемый метод находится в области видимости. Если вы хотите сослаться на статический метод, определенный в каком-либо другом классе, вам нужно будет указать его имя класса, как показано в листинге 9.6.

Делегаты не обязательно должны ссылаться на статические методы. Они могут ссылаться и на метод экземпляра. Есть несколько способов сделать это. Один из них — просто обратиться к методу экземпляра по имени из

контекста, в котором этот метод находится в области видимости. Метод `GetIsGreater Than Predicate` в листинге 9.7 возвращает делегат, который ссылается на `IsGreater Than`. Оба являются методами экземпляра, поэтому могут использоваться только со ссылкой на объект, но `GetIsGreater Than Predicate` имеет неявную ссылку `this`, и компилятор автоматически предоставляет ее делегату, который он неявно создает.

Листинг 9.6. Делегаты для методов в другом классе

```
internal class Program
{
    static void Main(string[] args)
    {
        Predicate<int> p1 = Tests.IsGreater ThanZero;
        Predicate<int> p2 = Tests.IsLess ThanZero;
    }
}

internal class Tests
{
    public static bool IsGreater ThanZero(int value) => value > 0;

    public static bool IsLess ThanZero(int value) => value < 0;
}
```

Листинг 9.7. Неявный делегат экземпляра

```
public class ThresholdComparer
{
    public int Threshold { get; set; }

    public bool IsGreater Than(int value) => value > Threshold;

    public Predicate<int> GetIsGreater Than Predicate() => IsGreater Than;
}
```

Кроме того, вы можете явно указать, какой экземпляр вам требуется. Листинг 9.8 создает три экземпляра класса `ThresholdComparer` из листинга 9.7, а затем создает три делегата, ссылающихся на метод `IsGreater Than`, по одному для каждого экземпляра.

Вам не нужно ограничивать себя простыми выражениями формы *переменная Name.MethodName*. Вы можете взять любое выражение, которое вычисляет ссылку на объект, а затем просто добавить `.MethodName`; если объект имеет

один или несколько методов с именем `MethodName`, это будет допустимое правило.

Листинг 9.8. Явный делегат экземпляра

```
var zeroThreshold = new ThresholdComparer { Threshold = 0 };
var tenThreshold = new ThresholdComparer { Threshold = 10 };
var hundredThreshold = new ThresholdComparer { Threshold = 100 };

Predicate<int> greaterThanZero = zeroThreshold.IsGreaterThan;
Predicate<int> greaterThanTen = tenThreshold.IsGreaterThan;
Predicate<int> greaterThanOneHundred = hundredThreshold.IsGreaterThan;
```



До сих пор я показывал только делегаты с одним аргументом, но можно определять типы делегатов с любым количеством аргументов. Например, библиотека классов определяет `Comparision<T>`, который сравнивает два элемента и, следовательно, принимает два аргумента (оба типа `T`).

C# не позволит вам создать делегат, который ссылается на метод экземпляра, не указав явно или неявно, какой именно экземпляр вы имеете в виду, и он всегда будет инициализировать делегат этим экземпляром.



Когда вы передаете делегат какому-либо другому коду, ему не нужно знать, является ли цель делегата статическим методом или методом экземпляра. И в случае методов экземпляра код, который использует делегат, не предоставляет этого экземпляра. Делегаты, которые ссылаются на методы экземпляров, всегда знают, к какому экземпляру они относятся, это же касается и метода.

Существует другой способ создания делегата, который может оказаться полезен, если вы до выполнения не знаете точно, какой метод или объект будете использовать: вы можете использовать API отражения (который я подробно объясню в главе 13). Сначала вы получаете `MethodInfo`, объект, представляющий определенный метод. Затем вы вызываете его метод `CreateDelegate`, передавая тип делегата и, при необходимости, целевой объект. (Если вы создаете делегат, ссылающийся на статический метод, целевой объект отсутствует, поэтому существует перегрузка, которая принимает только тип делегата.) Это создаст делегат со ссылкой на любой метод, на который

указывает экземпляр `MethodInfo`. Листинг 9.9 использует эту технику. Он получает объект `Type` (также часть API отражения; это способ ссылки на определенный тип), представляющий класс `ThresholdComparer`. Затем он запрашивает метод `MethodInfo`, представляющий метод `IsGreaterThan`. При этом он вызывает перегрузку `CreateDelegate`, которая принимает тип делегата и целевой экземпляра.

Листинг 9.9. `CreateDelegate`

```
MethodInfo m = typeof(ThresholdComparer).GetMethod("IsGreaterThan");
var greaterThanZero = (Predicate<int>) m.CreateDelegate(
    typeof(Predicate<int>), zeroThreshold);
```

Есть еще один способ выполнить ту же работу: тип `Delegate` имеет статический метод `CreateDelegate`, который устраниет необходимость запрашивать `MethodInfo`. Вы передаете ему два объекта типов — тип делегата и тип, определяющий целевой метод, а также имя метода. Если у вас уже есть `MethodInfo`, вы можете использовать его, но если у вас есть только имя, эта альтернатива более удобна.

Таким образом, делегат идентифицирует конкретную функцию, и если это функция экземпляра, делегат также содержит ссылку на объект. Но некоторые делегаты умеют еще больше.

Многоадресные делегаты

Если вы посмотрите на любой тип делегата с помощью инструмента обратного инжиниринга, такого как `ILDASM`, вы увидите, что независимо от того, является ли он типом, предоставляемым библиотекой классов .NET, или тем, который вы определили самостоятельно, он наследуется от базового типа, называемого `MulticastDelegate`¹. Как следует из названия, это означает, что делегаты могут ссылаться на более чем один метод. В основном это представляет интерес для сценариев, где вам может понадобиться вызвать несколько методов, чтобы уведомить о каком-то событии. Однако это поддерживают все делегаты независимо от того, нужно вам это или нет.

¹ `ILDASM` поставляется с Visual Studio. На момент написания статьи Microsoft не поддерживает кросс-платформенную версию, но доступны альтернативы с открытым исходным кодом.

Даже делегаты с непустыми типами возврата являются производными от `MulticastDelegate`. Обычно пользы от этого немного. Например, код, который требует `Predicate<T>` в обычных обстоятельствах будет проверять возвращаемое значение. `Array.FindIndex` использует это, чтобы узнать, соответствует ли элемент нашим критериям поиска. Если один делегат ссылается на несколько методов, что `FindIndex` делает с несколькими возвращаемыми значениями? Когда это происходит, будут выполняться все методы, но будут игнорироваться все возвращаемые значения, кроме возврата последнего выполняемого метода. (Как вы увидите в следующем разделе, это поведение по умолчанию, если вы не предоставляете специальную обработку для многоадресных делегатов.)

Функционал многоадресности доступен через статический метод `Combine` класса `Delegate`. Он принимает любые два делегата и возвращает один. Когда вызывается результирующий `pass[delegate]`, это выглядит так, как будто один за другим были вызваны два исходных делегата.

Это работает, даже когда передаваемые вами в `Combine` делегаты уже ссылаются на несколько методов, — собирая их вместе, можно создавать все более многоадресные делегаты. Если в обоих аргументах упоминается один и тот же метод, полученный объединенный делегат вызовет его дважды.



Метод `Combine` не изменяет ни один из передаваемых делегатов, но всегда создает новый.

Фактически же мы редко явно вызываем `Delegate.Combine`, потому что C# имеет встроенную поддержку объединения делегатов. Для этого вы можете использовать операторы `+` или `+=`. Листинг 9.10 демонстрирует оба варианта, объединяя три делегата из листинга 9.8 в один многоадресный делегат. Два полученных делегата эквивалентны — пример просто показывает два способа написания одного и того же. В обоих случаях результатом компиляции будет пара вызовов `Delegate.Combine`.

Вы также можете использовать операторы `-` или `-=` для создания нового делегата, который является копией первого операнда с удаленной последней ссылкой на метод, на который указывает второй операнд. Как вы можете догадаться, в данном случае используется `Delegate.Remove`.

Листинг 9.10. Объединение делегатов

```
Predicate<int> megaPredicate1 =  
    greaterThanZero + greaterThanTen + greaterThanOneHundred;  
  
Predicate<int> megaPredicate2 = greaterThanZero;  
megaPredicate2 += greaterThanTen;  
megaPredicate2 += greaterThanOneHundred;
```



Удаление делегата может удивить, если удаляемый вами делегат ссылается на несколько методов. Вычитание многоадресного делегата будет успешным, только если делегат, из которого оно производится, *последовательно и в том же порядке* содержит все методы вычитаемого делегата. (Операция, по сути, ищет единственное точное совпадение со своими входными данными, а не удаляет каждый содержащийся в них элемент.) В случае делегатов в листинге 9.10 вычитание (`greaterThanTen + greater ThanOneHundred`) из `megaPredicate1` будет работать, а вычитание (`greaterThanZero + greaterThanOneHundred`) — нет. Хотя `megaPredicate1` содержит ссылки на те же два метода и в том же порядке, последовательность другая, потому что у `megaPredicate1` есть дополнительный делегат в середине. Поэтому иногда проще избегать удаления многоадресных делегатов и удалять обработчики по одному, что позволяет избежать этих проблем.

Вызов делегата

До сих пор я показывал, как создавать делегат, но что, если вы пишете собственный API, который должен вызывать метод, предоставляемый вызывающей стороной? Во-первых, вам нужно будет выбрать тип делегата. Можно использовать предоставляемый библиотекой классов или определить свой собственный. Вы можете использовать этот тип делегата для параметра метода или свойства. В листинге 9.11 показано, что делать, если вам требуется вызвать метод (или методы), на который ссылается делегат.

Листинг 9.11. Вызов делегата

```
public static void CallMeRightBack(Predicate<int> userCallback)  
{  
    bool result = userCallback(42);  
    Console.WriteLine(result);  
}
```

Как показывает этот не очень реалистичный пример, вы можете использовать аргумент типа делегата, как если бы это была функция. Это будет

работать и для локальных переменных, полей и свойств. Фактически любое выражение, создающее делегат, может сопровождаться списком аргументов в скобках, а компилятор создаст код, который вызывает делегат. Если возвращает тип, отличный от `void`, значением выражения вызова будет то, что возвращает базовый метод (или, в случае делегата, ссылающегося на несколько методов, то, что возвращает финальный метод).

Хотя делегаты — это специальные типы, содержащие создаваемый во время выполнения код, в конечном счете в способе их вызова нет никакого волшебства. Вызов происходит в том же потоке, а исключения проходят через методы, которые были вызваны через делегата, точно так же, как если бы метод был вызван напрямую. Вызов делегата с одним целевым методом работает так, как если бы ваш код вызывал целевой метод обычным способом. Вызов многоадресного делегата аналогичен вызову каждого из его целевых методов по очереди.

Если вы хотите получить все возвращаемые значения из многоадресного делегата, вы можете взять процесс вызова в свои руки. Листинг 9.12 запрашивает список вызовов для делегата в виде массива, содержащего делегат с одним методом для каждого из методов, на которые ссылается исходный многоадресный делегат. Если исходный делегат содержал только один метод, этот список будет содержать только этот один делегат, но если используется функционал многоадресности, это дает возможность вызвать каждый из них по очереди. Таким образом можно увидеть, что содержит каждый отдельный предикат.



Листинг 9.12 основан на хитрости с `foreach`. Метод `GetInvocationList` возвращает массив типа `Delegate[]`. Цикл `foreach` тем не менее определяет тип переменной итерации как `Predicate<int>`. Это заставляет компилятор генерировать цикл, который по мере извлечения из коллекции приводит каждый элемент к этому типу. Это следует делать только в том случае, если вы уверены, что элементы относятся к этому типу, поскольку, если вы ошибаетесь, во время выполнения вас ждет исключение.

Есть еще один способ вызвать делегат, который иногда может оказаться полезен. Базовый класс `Delegate` содержит метод `DynamicInvoke`. Вы можете вызвать его для делегата любого типа во время компиляции, не зная точно, какие аргументы ему требуются. Он принимает массив `params` типа `object[]`,

поэтому вы можете передать любое количество аргументов. Он проверит количество и тип аргументов уже во время выполнения. Это можно задействовать в некоторых сценариях позднего связывания, хотя встроенные динамические функции (обсуждаемые в главе 2), добавленные в C# 4.0, являются более всеобъемлющими. Однако ключевое слово `dynamic` немного тяжеловеснее из-за своей излишней гибкости, поэтому если `DynamicInvoke` делает именно то, что вам нужно, это лучший выбор.

Листинг 9.12. Вызов каждого делегата по отдельности

```
public static void TestForMajority(Predicate<int> userCallbacks)
{
    int trueCount = 0;
    int falseCount = 0;
    foreach (Predicate<int> p in userCallbacks.GetInvocationList())
    {
        bool result = p(42);
        if (result)
        {
            trueCount += 1;
        }
        else
        {
            falseCount += 1;
        }
    }
    if (trueCount > falseCount)
    {
        Console.WriteLine("The majority returned true");
    }
    else if (falseCount > trueCount)
    {
        Console.WriteLine("The majority returned false");
    }
    else
    {
        Console.WriteLine("It's a tie");
    }
}
```

Распространенные типы делегатов

Библиотека классов .NET содержит несколько полезных типов делегатов, и во многих ситуациях вы можете использовать именно их, вместо того что-

бы определять свои собственные. Например, библиотека классов определяет набор обобщенных делегатов `Action` с различным количеством параметров типа. Все они следуют общему шаблону: для каждого параметра типа есть единственный параметр метода этого типа. В листинге 9.13 показаны первые четыре, включая вариант с нулевым аргументом.

Листинг 9.13. Первые несколько делегатов `Action`

```
public delegate void Action();
public delegate void Action<in T1>(T1 arg1);
public delegate void Action<in T1, in T2 >(T1 arg1, T2 arg2);
public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
```

Хотя это явно открытая концепция — вполне можно вообразить себе такой делегат с любым количеством аргументов — CTS не позволяет определить подобный тип как шаблон, поэтому библиотека классов должна определять каждую форму как отдельный тип. Следовательно, не существует варианта `Action` с 200 аргументами. Верхний предел составляет 16 аргументов.

Очевидное ограничение `Action` состоит в том, что эти типы имеют возвращаемый тип `void` и поэтому не могут ссылаться на методы, которые возвращают значения. Но есть похожее семейство типов делегатов, `Func`, которое допускает любой тип возврата. В листинге 9.14 показаны первые несколько делегатов этого семейства, и, как вы можете видеть, они очень похожи на `Action`. Они просто содержат дополнительный конечный параметр типа, `TResult`, который указывает тип возвращаемого значения. Как и с `Action<T>`, вы можете дойти до 16 аргументов.

Листинг 9.14. Первые несколько делегатов `Func`

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg1);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<in T1, in T2, in T3, out TResult>(
    T1 arg1, T2 arg2, T3 arg3);
```

Скорее всего, эти два семейства делегатов удовлетворят большинству требований. Если вы не пишете исполинские методы с более чем 16 аргументами, зачем вам вообще что-то еще? Почему библиотека классов определяет отдельный `Predicate<T>` когда вместо этого можно просто использовать `Func<T, bool>?` В некоторых случаях ответ кроется в истории: многие типы

делегатов существовали еще до того, как были добавлены эти типы общего назначения. Но это не единственная причина — новые типы делегатов продолжают добавляться даже сейчас. Основная причина в том, что иногда бывает полезно определить специализированный тип делегата для указания конкретной семантики. Также, если вам нужен делегат, который способен работать с аргументами `ref` или `out`, вам придется написать соответствующий тип делегата.

Если у вас есть `Func<T, bool>`, вы знаете лишь то, что у вас есть метод, который принимает `T` и возвращает `bool`. Но в случае `Predicate<T>` уже имеется подразумеваемое значение: он принимает решение об экземпляре `T` и возвращает `true` или `false` соответственно; не все методы, которые принимают один аргумент и возвращают `bool`, обязательно соответствуют этому шаблону. Предоставляя `Predicate<T>`, вы не просто говорите, что у вас есть метод с определенной сигнатурой, вы говорите, что у вас есть метод, который служит определенной цели. Например, `HashSet<T>` (описан в главе 5) содержит метод `Add`, который принимает один аргумент и возвращает `bool`, поэтому он соответствует сигнатуре `Predicate <T>`, но не тождествен ему семантически. Основная задача `Add` — выполнять действие с побочными эффектами, возвращая некоторую информацию о проделанном действии, тогда как предикаты просто сообщают вам что-то о значении или объекте. (Как это бывает, `Predicate<T>` появился до `Func<T, bool>`, так что история является одной из причин, по которой некоторые API его используют. Тем не менее семантика все еще имеет значение — есть некоторые более новые API, для которых `Func<T, bool>` был хорошим вариантом, но они тем не менее используют `Predicate<T>`.)

Поскольку в C# 7 появились типы `ref struct`, появилась еще одна причина для определения пользовательского типа делегата: вы не можете использовать `ref struct` в качестве аргумента обобщенного типа. (Глава 18 посвящена этим типам.) Так что если вы попытаетесь создать обобщенный тип `Action<T>` со `Span<int>`, который является типом `ref struct`, написав `Action<Span<int>>`, вы получите ошибку компилятора. Причиной этого является то, что типы `ref struct` могут использоваться только в определенных сценариях (они всегда должны находиться в стеке) и нет никакого способа определить, использует ли конкретный обобщенный тип или метод свои аргументы типа только разрешенными способами. (Можно представить себе направленный на это новый тип ограничения аргумента типа, но на момент написания такого ограничения пока не существует.) Поэтому если вам нужен тип делегата,

который может ссылаться на метод, принимающий аргумент `ref struct`, он должен быть специально созданным, необобщенным делегатом.

Библиотека классов .NET определяет множество типов делегатов, большинство из которых еще более специализированы, чем `Predicate<T>`. Например, пространство имен `System.IO` и его потомки определяют несколько типов, относящихся к определенным событиям, например `SerialPinChangedEventHandler`, который используется только при работе со старомодными последовательными портами, такими как некогда распространенный интерфейс `RS232`.

Совместимость типов

Типы делегатов не являются производными друг от друга. Любой тип делегата, определенный вами в C#, будет напрямую происходить от `MulticastDelegate`, как и все типы делегатов в библиотеке классов. Однако система типов поддерживает определенные неявные ссылочные преобразования для обобщенных типов делегатов посредством ковариантности и контравариантности. Правила очень похожи на таковые для интерфейсов. Как показывает ключевое слово `in` в листинге 9.3, аргумент типа `T` в `Predicate<T>` является контравариантным, что означает, что если между типами `A` и `B` существует неявное ссылочное преобразование, то оно также существует между типами `Predicate` и `Predicate<A>`. В листинге 9.15 показано такое неявное преобразование.

Листинг 9.15. Ковариантность делегатов

```
public static bool IsLongString(object o)
{
    return o is string s && s.Length > 20;
}

static void Main(string[] args)
{
    Predicate<object> po = IsLongString;
    Predicate<string> ps = po;
    Console.WriteLine(ps("Too short"));
}
```

Сначала метод `Main` создает `Predicate<object>`, ссылаясь на метод `IsLongString`. Любой целевой метод для этого типа предиката способен проверять любой объект любого типа; таким образом, он явно способен удовлетворить

потребностям кода, для которого требуется предикат, способный проверять строки, поэтому вполне разумно, что преобразование в `Predicate<string>` благодаря контравариантности оказывается успешным. Ковариантность работает так же, как и с интерфейсами, поэтому обычно она связана с типом возврата делегата. (Обозначим параметры ковариантного типа ключевым словом `out`.) Все встроенные типы делегатов `Func` имеют аргумент ковариантного типа, представляющий тип возврата функции и называемый `TResult`. Все параметры типа для параметров функции являются контравариантными, как и все аргументы типа для типов делегатов `Action`.



Преобразования делегатов на основе варианты являются неявными ссылочными преобразованиями. Это означает, что при преобразовании ссылки результат все равно ссылается на тот же экземпляр делегата. (Этим характеризуются все неявные преобразования ссылок, но не все неявные преобразования работают таким образом. Неявные числовые преобразования создают новый экземпляр целевого типа; неявные преобразования упаковки создают новый блок в куче.) Таким образом, в листинге 9.15 `ro` и `ps`зываются на один и тот же делегат в куче. Это немного отличается от присвоения `IsLongString` обеим переменным, что приведет к созданию двух делегатов разных типов.

Вы также можете ожидать, что одинаково выглядящие делегаты окажутся совместимы. Например, `Predicate<int>` может ссылаться на любой метод, который может использовать `Func<int, bool>`, и наоборот, поэтому вполне ожидаемо неявное преобразование между этими двумя типами. Возможно, вас еще больше вдохновит раздел «Совместимость делегатов» в спецификации C#, в котором говорится, что делегаты с одинаковыми списками параметров и типами возвращаемых данных тоже совместимы. (На самом деле он идет еще дальше, говоря, что допускаются даже некоторые различия. Например, я упоминал ранее, что типы аргументов могут отличаться, если доступны определенные неявные ссылочные преобразования.) Но если вы попробуете запустить код листинга 9.16, он не будет работать.

Листинг 9.16. Недопустимое преобразование делегата

```
Predicate<string> pred = IsLongString;
Func<string, bool> f = pred; // Завершится с ошибкой компилятора
```

Явное приведение также не сработает — добавление оного исключит ошибку компилятора, но вместо нее вы получите ошибку времени выполнения.

Стандартная система типов считает, что это несовместимые типы, поэтому переменная, объявленная с одним типом делегата, не может содержать ссылку на другой, даже если их сигнатуры метода совместимы (за исключением случаев, когда два рассматриваемых типа делегата основаны на одном и том же типе делегата и совместимы за счет ковариантности или контравариантности). Это не тот сценарий, для которого разработаны правила совместимости делегатов C# — в основном они используются для выяснения, может ли конкретный метод быть целью для конкретного типа делегата.

Отсутствие совместимости типов между «совместимыми» типами делегатов может показаться странным, но структурно идентичные типы делегатов не обязательно имеют одинаковую семантику, как мы уже видели на примере `Predicate<T>` и `Func<T, bool>`. Если вам нужно выполнить такое преобразование, это может быть признаком того, что с дизайном вашего кода что-то не так².

Тем не менее можно создать новый делегат, который ссылается на тот же метод, что и оригинал, если новый тип совместим со старым. Всегда полезно остановиться и спросить себя, зачем это нужно. Тем не менее иногда это необходимо и на первый взгляд выглядит просто. Листинг 9.17 показывает один из способов. Однако, как показывает оставшаяся часть этого раздела, все немного сложнее, чем кажется. Кроме того, это не самое эффективное решение (еще одна причина, по которой вам стоит поразмыслить над изменением дизайна кода, чтобы изначально избежать необходимости делать это).

Листинг 9.17. Делегат, ссылающийся на другого делегата

```
Predicate<string> pred = IsLongString;
var pred2 = new Func<string, bool>(pred); // Менее эффективен, чем
прямая ссылка
```

Проблема с листингом 9.17 заключается в том, что он добавляет ненужный уровень косвенности. Второй делегат не ссылается на тот же метод, что и первый. В действительности он ссылается на первый делегат — поэтому, вместо делегата, являющегося ссылкой на `IsLongString`, переменная `pred2` содержит ссылку на делегат, который является ссылкой на делегат, являющийся ссылкой на `IsLongString`. Это потому, что компилятор обрабатывает листинг 9.17 так, как будто вы написали код листинга 9.18. (Все типы делегатов имеют

² Или вы можете просто быть одним из энтузиастов языка с аллергией на выражение семантики через статические типы. Если это так, C# может оказаться не вашим языком.

метод `Invoke`. Он реализован CLR и выполняет работу, необходимую для вызова всех методов, на которые ссылается делегат.)

Листинг 9.18. Делегат, явно ссылающийся на другой делегат

```
Predicate<string> pred = IsLongString;
var pred2 = new Func<string, bool>(pred.Invoke);
```

В листингах 9.17 и 9.18, когда вы вызываете второй делегат через переменную `pred2`, он, в свою очередь, вызывает делегат, на который ссылается `pred`, что в итоге приводит к вызову метода `IsLongString`. Метод правильный, но вызывается он не так непосредственно, как хотелось бы. Если вы знаете, что делегат ссылается на единственный метод (т. е. вы не используете функционал многоадресности), листинг 9.19 дает более прямолинейный результат.

Листинг 9.19. Новый делегат для текущей цели

```
Predicate<string> pred = IsLongString;
var pred2 = (Func<string, bool>) pred.Method.CreateDelegate(
    typeof(Func<string, bool>), pred.Target);
```

Из делегата `pred` извлекается свойство `MethodInfo`, представляющее целевой метод, и ему передается требуемый тип делегата, а также цель делегата `pred`, после чего создается новый делегат `Func<string, bool>`. Результатом становится новый делегат, который непосредственно ссылается на тот же метод `IsLongString`, что и `pred`. (`Target` будет содержать `null`, потому что это статический метод, но я передаю его в `CreateDelegate`, потому что хочу продемонстрировать код, который подходит как для статических методов, так и для методов экземпляра.) Если вы имеете дело с многоадресными делегатами, то листинг 9.19 не будет работать, так как предполагает, что есть только один целевой метод. Вам нужно будет аналогичным образом вызывать `CreateDelegate` для каждого элемента в списке вызовов. Это не то, что можно увидеть очень часто, но для полноты картины листинг 9.20 показывает, как это делается.



В листинге 9.20 аргумент для параметра типа `TResult` должен быть делегатом, поэтому я задал для него соответствующее ограничение. Обратите внимание, что C# не позволял задавать такого рода ограничения до C# 7.3, поэтому если вы видите код, который, казалось бы, должен содержать такое ограничение, но не содержит, это может быть причиной.

Листинг 9.20. Преобразование многоадресного делегата

```
public static TResult DuplicateDelegateAs<TResult>(MulticastDelegate
    source)
    where TResult : Delegate
{
    Delegate result = null;
    foreach (Delegate sourceItem in source.GetInvocationList())
    {
        var copy = sourceItem.Method.CreateDelegate(
            typeof(TResult), sourceItem.Target);
        result = Delegate.Combine(result, copy);
    }

    return (TResult) (object) result;
}
```

Последние несколько примеров зависели от различных членов типов делегатов: `Target`, `Method` и `Invoke`. Первые два происходят из класса `Delegate`, являющийся базовым классом для `MulticastDelegate`, из которого наследуются все типы делегатов. Тип свойства `Target` — это `object`. Свойство будет содержать `null`, если делегат ссылается на статический метод; в противном случае он будет ссылаться на экземпляр, для которого вызван метод. Свойство `Method` возвращает `MethodInfo`, идентифицирующий целевой метод. Третий элемент, `Invoke`, создается компилятором. Это один из немногих стандартных членов, которые компилятор C# создает при определении типа делегата.

За кулисами синтаксиса

Хотя для определения типа делегата требуется всего одна строка кода (как показано в листинге 9.3), компилятор превращает его в тип, который содержит три метода и конструктор. И конечно же, тип наследует члены своих базовых классов. Все делегаты являются производными от `MulticastDelegate`, хотя все важные члены экземпляра родом из его базового класса `Delegate`. (`Delegate` происходит от `object`, поэтому у всех делегатов есть общие методы объекта.) Даже `GetInvocationList`, явно ориентированный на многоадресность, определяется базовым классом `Delegate`.

Я уже показал все открытые члены экземпляра, которые определяет `Delegate`. `DynamicInvoke`, `GetInvocationList`, `Target` и `Method`. В листинге 9.21 показаны сигнатуры созданного компилятором конструктора и методов для типа

делегата. Детали варыируют от типа к типу; конкретно эти созданы для типа `Predicate<T>`.

Листинг 9.21. Члены типа делегата

```
public Predicate(object target, IntPtr method);  
  
public bool Invoke(T obj);  
  
public IAsyncResult BeginInvoke(T obj, AsyncCallback callback,  
                               object state);  
public bool EndInvoke(IAsyncResult result);
```



Разделение между `Delegate` и `MulticastDelegate` является бессмысленным и случайным результатом исторического казуса. Первоначально планировалось поддерживать как многоадресные, так и однoadресные делегаты, но к концу периода предварительного выпуска для .NET 1.0 об этом различии забыли, и теперь все типы делегатов поддерживают многоадресные экземпляры. Это произошло достаточно поздно, так что Microsoft почувствовала, что объединять два базовых типа в один довольно рискованно, поэтому разделение осталось, пусть даже и бессмысленное.

Любой тип делегата, который вы определите, будет иметь четыре одинаковых члена, и ни один из них не будет иметь тела. Компилятор создает объявления, а CLR во время выполнения автоматически добавляет реализацию.

Конструктор принимает целевой объект (`null` для статических методов) и `IntPtr`, идентифицирующий метод. Обратите внимание, что это не `MethodInfo`, который возвращает свойство `Method`, а признак функции, нечеткий двоичный идентификатор целевого метода³. CLR может предоставлять двоичные признаки метаданных для всех членов и типов, но в C# нет синтаксиса для работы с ними, поэтому обычно мы с ними не сталкиваемся. Когда вы создаете новый экземпляр типа делегата, компилятор автоматически создает промежуточный язык, который получает признак функции.

³ `IntPtr` — тип значения, обычно используемый для непрозрачных значений дескриптора. Иногда это можно встретить в сценариях взаимодействия — в тех редких случаях, когда вы видите в .NET необработанный дескриптор из API ОС, он может быть представлен как `IntPtr`, хотя во многих случаях заменяется на `SafeHandle`.

Причина, по которой делегаты для внутренней работы используют признаки, заключается в том, что иногда они оказываются более эффективными, чем работа с типами API отражения, такими как `MethodInfo`.

Метод `Invoke` вызывает целевой метод делегата (или методы), и в C# вы можете использовать его явно, как показано в листинге 9.22. Он почти идентичен листингу 9.11, с той лишь разницей, что за переменной делегата следует `.Invoke`. В результате создается точно такой же код, как в листинге 9.11. Используете вы `Invoke` или просто синтаксис, который обрабатывает идентификаторы делегатов, как если бы они были именами методов, — это лишь вопрос стиля. Как бывший разработчик C++, я всегда чувствовал себя непринужденно, работая с синтаксисом листинга 9.11. Он похож на использование указателей на функции в этом языке, но есть мнение, что явное написание `Invoke` облегчает понимание того, что код использует делегат.

Листинг 9.22. Явное использование `Invoke`

```
public static void CallMeRightBack(Predicate<int> userCallback)
{
    bool result = userCallback.Invoke(42);
    Console.WriteLine(result);
}
```

Одним из преимуществ явной формы является то, что вы можете использовать `null`-условный оператор для случая, когда переменная-делегат равна `null`. Листинг 9.23 осуществляет попытку вызова только в том случае, если указанный аргумент не равен `null`.

Листинг 9.23. Использование `Invoke` с `null`-условным оператором

```
public static void CallMeMaybe(Action<int> userCallback)
{
    userCallback?.Invoke(42);
}
```

Метод `Invoke` — это вместилище сигнатуры метода типа делегата. Когда вы определяете тип делегата, тип возвращаемого значения и список параметров попадают именно сюда. Когда компилятору необходимо проверить, совместим ли конкретный метод с типом делегата (например, при создании нового делегата этого типа), компилятор сравнивает метод `Invoke` с предоставленным вами методом.

Как показано в листинге 9.21, все типы делегатов также имеют методы `BeginInvoke` и `EndInvoke`. Они считаются устаревшими и не работают в .NET Core. (При этом выдают исключение `PlatformNotSupportedException`.) Они по-прежнему работают в .NET Framework, но они действительно устарели. Их задача — обеспечение асинхронного вызова методов через пул потоков, что называется *асинхронным вызовом делегата*. Когда-то это был популярный способ выполнения асинхронной работы, но в ранних версиях .NET он перестал широко использоваться незадолго до того, как стал устаревшим, и для этого есть три причины. Во-первых, в .NET 4.0 была представлена библиотека параллельных задач (TPL), которая обеспечивает более гибкую и мощную абстракцию для служб пула потоков. (Подробнее см. в главе 16.) Во-вторых, эти методы реализуют более старый шаблон, известный как модель асинхронного программирования (также описанная в главе 16), который напрямую не соответствует новым функциям асинхронного языка C# (описаны в главе 17). Наконец, наибольшее преимущество асинхронного вызова делегата заключается в том, что он обеспечивает простой способ передачи набора значений из одного потока в другой — вы можете просто передать все, что вам нужно, в качестве аргументов делегата. Однако в C# 2.0 появился гораздо лучший способ решения проблемы: анонимные функции.

Анонимные функции

C# позволяет создавать делегаты без необходимости явно определять отдельный метод. Вы можете написать специальный вид выражения, значением которого является метод. Вы можете думать о них как о *методах-выражениях* или *функциях-выражениях*, но официальное название — *анонимные функции*. Выражения могут быть непосредственно переданы в качестве аргументов или присвоены переменным, поэтому методы, создаваемые этими выражениями, не имеют имен. (По крайней мере, не в C#. Среда выполнения требует, чтобы все методы имели имена, поэтому C# генерирует для них скрытые имена, но с точки зрения языка C# они все-таки являются анонимными.)

Для простых методов, возможность записывать их в виде выражений может серьезно уменьшить беспорядок. И как мы увидим в разделе «Захват переменных» на с. 495, компилятор использует тот факт, что делегат — это не просто ссылка на метод, призванная предоставить анонимным функциям

доступ к любым переменным, находящимся в области видимости в содержащем методе в момент, когда появляется анонимная функция.

По историческим причинам C# предоставляет два способа определения анонимной функции. Более старый способ, включающий применение ключевого слова `delegate`, показан в листинге 9.24. Эта форма известна как анонимный метод. Я поместил каждый аргумент `FindIndex` в отдельной строке, чтобы выделить анонимные функции (второй аргумент), но C# этого не требует⁴.

Листинг 9.24. Синтаксис анонимного метода

```
public static int GetIndexOffFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(
        bins,
        delegate (int value) { return value > 0; }
    );
}
```

В некотором смысле это напоминает обычный синтаксис определения методов. Список параметров отображается в скобках и сопровождается блоком, содержащим тело метода (который, кстати, может содержать столько кода, сколько вам нужно, а также может содержать вложенные блоки, локальные переменные, циклы и все, что можно поместить в обычный метод). Но вместо имени метода у нас просто ключевое слово `delegate`. Компилятор сам выводит тип возвращаемого значения. В этом случае сигнатурой метода `FindIndex` объявляется второй аргумент как `Predicate<T>`, что сообщает компилятору, что тип возвращаемого значения должен быть `bool`.

Вообще, компилятору известно больше. Я передал `FindIndex` массив `int[]`, поэтому компилятор знает, что аргумент типа `T` — это `int`, поэтому нам нужен `Predicate<int>`. Это означает, что в листинге 9.24 мне пришлось предоставить информацию — тип аргумента делегата, — которую компилятор уже знал. В C# 3.0 введен более компактный синтаксис анонимных функций,

⁴ Очень неудобно, когда есть два похожих термина, которые без достаточных оснований означают практически одно и то же. Документация C# использует термин «анонимная функция» в качестве общего термина для любого вида выражения метода. Анонимный метод был бы лучшим названием для этого, потому что не все эти сущности являются функциями в строгом смысле слова. Они могут содержать `void` в качестве возвращаемого значения, но к тому времени, когда Microsoft понадобился общий термин для подобных вещей, это имя уже было занято.

который лучше задействует возможности компилятора делать выводы. Он показан в листинге 9.25.

Листинг 9.25. Лямбда-синтаксис

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(
        bins,
        value => value > 0
    );
}
```

Эта форма анонимной функции называется *лямбда-выражением*, и название позаимствовано из теории вычислений. Особого значения в выборе греческой буквы «лямбда» (λ) нет. Это случайный результат ограничений технологии печати 1930-х годов. Изобретатель лямбда-исчисления Алонзо Черч изначально хотел использовать другое обозначение, но когда публиковал свою первую статью на эту тему, оператор печатного станка решил напечатать λ , поскольку эта буква больше всего напоминала условный знак Черча (из всего того, что станок мог воспроизвести). Несмотря на случайное происхождение, этот произвольно выбранный термин стал вездесущим. LISP, ранний и влиятельный язык программирования, использовал имя *лямбда* для выражений, которые являются функциями, и с тех пор многие языки, включая C#, последовали его примеру.

Листинг 9.25 в точности соответствует листингу 9.24; мне лишь удалось исключить несколько вещей. Символ `=>` однозначно указывает на лямбду, поэтому компилятору не нужно громоздкое и уродливое ключевое слово `delegate`, чтобы распознать анонимную функцию. Из окружающего контекста компилятор знает, что метод должен принимать `int`, поэтому в указании типа параметра нет необходимости; хватило лишь имени параметра: `value`. Для простых методов, которые состоят только из одного выражения, лямбда-синтаксис позволяет опустить блок и оператор `return`. Все это позволяет писать очень компактные лямбды, но в некоторых случаях вам захочется что-то сохранить, поэтому, как показано в листинге 9.26, существуют различные дополнительные возможности. Каждая лямбда в этом примере эквивалентна.

Первый вариант заключается в том, что вы можете заключить параметр в круглые скобки. Это необязательно в случае с одним параметром, но обязательно для лямбды с несколькими параметрами. Вы также можете явно

указать типы параметров (в этом случае вам также понадобятся скобки, даже если есть только один параметр). И, если хотите, вы можете использовать блок вместо единого выражения, после чего вам также придется использовать ключевое слово `return`, если лямбда-выражение возвращает значение. Обычная причина использования блока — желание записать несколько операторов внутри метода.

Листинг 9.26. Варианты лямбда-выражений

```
Predicate<int> p1 = value => value > 0;
Predicate<int> p2 = (value) => value > 0;
Predicate<int> p3 = (int value) => value > 0;
Predicate<int> p4 = value => { return value > 0; };
Predicate<int> p5 = (value) => { return value > 0; };
Predicate<int> p6 = (int value) => { return value > 0; };
```

Вы можете задаться вопросом, а зачем вообще нужно так много различных форм? Почему бы раз и навсегда не утвердить единый синтаксис? Хотя последняя строка листинга 9.26 показывает наиболее общую форму, она же гораздо более громоздкая по сравнению с первой. Поскольку одна из целей лямбда-выражений — предоставить более краткую альтернативу анонимным методам, C# поддерживает более короткие формы там, где их можно использовать без двусмысленности.

Вы также можете написать лямбду, которая не принимает аргументов. Как показано в листинге 9.27, можно просто поставить пустую пару скобок перед символом `=>`. (И как показывает этот пример, лямбда-выражения, использующие оператор больше или равно, `>=`, могут выглядеть немного странно из-за не имеющего смысла сходства между символами `=>` и `>=`.)

Листинг 9.27. Лямбда с нулевым аргументом

```
Func<bool> isAfternoon = () => DateTime.Now.Hour >= 12;
```

Гибкий и компактный синтаксис привел к тому, что лямбды почти вытеснили старый синтаксис анонимного метода. Однако в старом синтаксисе содержится одно преимущество: он позволяет полностью опустить список аргументов. В некоторых ситуациях, когда вы предоставляете обратный вызов, вам необходимо знать лишь то, что ожидаемое событие уже произошло.

Это особенно часто встречается при использовании стандартного шаблона событий, описанного далее в этой главе, потому что для него требуется, чтобы обработчики событий принимали аргументы даже в ситуациях, когда они

не нужны. Например, при нажатии кнопки мало что можно сказать, кроме того, что она была нажата, но все же все типы кнопок в различных каркасах разработки интерфейса пользователя .NET передают обработчику событий два аргумента. Листинг 9.28 благополучно это игнорирует, используя анонимный метод, который пропускает список параметров.

Листинг 9.28. Игнорирование аргументов в анонимном методе

```
EventHandler clickHandler = delegate { Debug.WriteLine("Clicked!"); };
```

EventHandler — это тип делегата, который требует, чтобы его целевые методы принимали два аргумента, типа `object` и `EventArgs`. Если бы нашему обработчику требовался доступ к какому-либо из них, мы могли бы, конечно, добавить список параметров, но синтаксис анонимного метода позволяет нам при желании полностью его опустить. С лямбдой этого сделать не получится.

Захват переменных

Хотя анонимные функции зачастую занимают гораздо меньше места в исходном коде, нежели полноценный обычный метод, их преимущества не сводятся лишь к лаконичности. Компилятор C# использует способность делегата ссылаться не только на метод, но и на некоторый дополнительный контекст, что дает нам крайне полезный функционал: делегат может сделать переменные из содержащего метода доступными для анонимной функции. В листинге 9.29 показан метод, который возвращает `Predicate<int>`. Он создает его с помощью лямбды, которая использует аргумент из содержащего метода.

Листинг 9.29. Использование переменной из содержащего метода

```
public static Predicate<int> IsGreaterThan(int threshold)
{
    return value => value > threshold;
}
```

Это обеспечивает те же функциональные возможности, что и класс `ThresholdComparer` из листинга 9.7, но теперь цель достигается одним простым методом, не требующим от нас написания целого класса. Мы можем записать все еще компактнее, если вместо тела метода используем выражение, как это показано в листинге 9.30.

Листинг 9.30. Использование переменной из содержащего метода

```
public static Predicate<int> IsGreaterThan(int threshold) =>
    value => value > threshold;
```

Этот код может легко обмануть своей простотой, поэтому стоит внимательно присмотреться к тому, что он делает. Метод `IsGreaterThan` возвращает экземпляр делегата. Целевой метод этого делегата выполняет простое сравнение — он вычисляет выражение `value > threshold` и возвращает результат. Переменная `value` в этом выражении является просто аргументом делегата. Это `int`, передаваемый любым кодом, вызывающим `Predicate<int>`, который возвращает `IsGreaterThan`. Вторая строка листинга 9.31 вызывает этот код, передавая значение 200 в качестве аргумента для `value`.

Листинг 9.31. Откуда берется `value`

```
Predicate<int> greaterThanTen = IsGreaterThan(10);
bool result = greaterThanTen(200);
```

С переменной `threshold` в выражении все немного сложнее. Это не аргумент анонимной функции. Это аргумент `IsGreaterThan`, и в листинге 9.31 в качестве аргумента `threshold` передается значение 10. Однако `IsGreaterThan` должен завершиться, прежде чем мы сможем вызвать делегат, который он возвращает. Поскольку метод, для которого переменная `threshold` является аргументом, уже завершен, вы можете решить, что она перестанет быть доступной ко времени вызова делегата. На самом деле все хорошо, потому что компилятор выполнит кое-какие действия от нашего имени. Если анонимная функция использует аргументы или локальные переменные, которые были объявлены содержащим методом, компилятор создает класс для хранения этих переменных, чтобы они могли пережить метод, который их создал. Для создания экземпляра этого класса компилятор добавляет код в содержащий метод. (Вспомните, что каждый вызов блока получает собственный набор локальных переменных, поэтому, если какие-либо локальные объекты помещаются в объект с целью продления их времени жизни, для каждого вызова потребуется новый объект.) Это одна из причин, почему ошибочен популярный миф о том, что локальные переменные значимых типов всегда находятся в стеке, — в нашем случае компилятор копирует значение входящего аргумента `threshold` в поле объекта в куче и код, использующий переменную `threshold`, в конечном итоге использует вместо нее это поле. В листинге 9.32 показан код, который компилятор создает для анонимной функции в листинге 9.29.

Имена классов и методов начинаются с символов, которые недопустимы в идентификаторах C#, для того чтобы гарантировать, что этот созданный компилятором код не будет конфликтовать с кодом, который пишем мы. Технически это непередаваемое имя. (Между прочим, точные имена не являются фиксированными — запустив пример, вы можете обнаружить, что они немногого отличаются.) Этот созданный код имеет поразительное сходство с классом `ThresholdComparer` из листинга 9.7, что неудивительно, поскольку цель та же самая: делегату нужен какой-то метод, на который он может ссылаться, и поведение этого метода зависит от значения, которое не является фиксированным. Анонимные функции не являются особенностью системы типов среды выполнения, поэтому компилятор вынужден создавать класс для обеспечения такого рода поведения поверх базовой функциональности делегата CLR.

Листинг 9.32. Код, созданный для анонимной функции

```
[CompilerGenerated]
private sealed class <>c__DisplayClass1_0
{
    public int threshold;

    public bool <IsGreaterThan>b__0(int value)
    {
        return (value > this.threshold);
    }
}
```



Локальные функции (описанные в главе 3) также могут обращаться к локальным переменным своих содержащих методов. Как правило, это не меняет время жизни этих переменных, потому что локальная функция недоступна вне содержащего ее метода. Но если вы создаете делегат, который ссылается на локальную функцию, это означает, что он может быть вызван после возврата содержащего метода. В этом случае компилятор проделает тот же трюк, что и в случае анонимных функций, позволяя переменным оставаться действительными и после завершения внешнего метода.

Если вы знаете, что в действительности происходит, когда вы пишете анонимную функцию, для вас будет естественным следствием тот факт, что внутренний метод может не только читать переменную, но и изменять ее. Эта переменная — просто поле в объекте, к которому имеют доступ два метода:

анонимная функция и содержащий метод. Именно так в листинге 9.33 осуществляется работа счетчика, который обновляется из анонимной функции.

Листинг 9.33. Изменение захваченной переменной

```
static void Calculate(int[] nums)
{
    int zeroCount = 0;
    int[] nonZeroNums = Array.FindAll(
        nums,
        v =>
    {
        if (v == 0)
        {
            zeroCount += 1;
            return false;
        }
        else
        {
            return true;
        }
    });
    Console.WriteLine($"Number of zero entries: {zeroCount}");
    Console.WriteLine($"First non-zero entry: {nonZeroNums[0]}");
}
```

Все в области видимости содержащего метода также находится в области видимости анонимных функций. Если содержащий метод является методом экземпляра, то это включает в себя любые элементы экземпляра типа, поэтому ваша анонимная функция может обращаться к полям, свойствам и методам. (Компилятор добивается этого, добавляя поле в созданный класс для хранения копии ссылки `this`.) В созданные классы, подобные показанному в листинге 9.32, компилятор помещает только то, что ему нужно, и, если вы не используйте переменные или члены экземпляра из содержащей области, ему может вообще не потребоваться создавать класс, и вместо этого он просто добавит статический метод к существующему типу.

Метод `FindAll` в предыдущих примерах не хранит делегат после возврата — любые обратные вызовы будут происходить во время выполнения `FindAll`. Однако так работает далеко не все. Некоторые API работают асинхронно и осуществляют обратный вызов в какой-то момент в будущем, а к тому времени содержащийся метод может уже завершиться. Это означает, что любые переменные, захваченные анонимной функцией, переживут содержащий

метод. В общем, это нормально, потому что все захваченные переменные содержатся в объекте в куче, так что анонимная функция не полагается на фрейм стека, которого больше нет. Единственное, о чем необходимо заботиться, так это о явном освобождении ресурсов до завершения обратных вызовов. Листинг 9.34 показывает ошибку, которую очень легко допустить. При этом используется асинхронный API на основе обратного вызова для загрузки ресурса по определенному URL-адресу через HTTP. (Листинг вызывает метод `ContinueWith` метода `Task<Stream>`, возвращаемого `HttpClient.GetStreamAsync`, и передает делегат, который будет вызван после получения ответа HTTP.)

Этот метод является частью параллельной библиотеки задач, описанной в главе 16.

Листинг 9.34. Преждевременная очистка

```
HttpClient http = GetHttpClient();
using (FileStream file = File.OpenWrite(@"c:\temp\page.txt"))
{
    http.GetStreamAsync("https://endjin.com/")
        .ContinueWith((Task<Stream> t) => t.Result.CopyToAsync(file));
} // Скорее всего, StreamWriter будет удален до запуска обратного вызова
```

Оператор `using` в этом примере удалит `FileStream`, как только выполнение достигнет точки, в которой переменная `file` выходит из области видимости во внешнем методе. Проблема в том, что переменная `file` также используется в анонимной функции, которая, по всей вероятности, будет запущена после того, как поток, выполняющий этот внешний метод, выйдет из блока оператора `using`. Компилятор понятия не имеет, когда начнет работать внутренний блок, — он не знает, является ли это синхронным обратным вызовом, как в случае с `Array.FindAll`, или же асинхронным. Так что здесь он не будет делать ничего особенного — он просто вызывает `Dispose` в конце блока, как и велит наш код.



Функционал асинхронного языка, обсуждаемый в главе 17, может помочь избежать такого рода проблем. Когда вы применяете их для использования API, которые следуют этому типу шаблона на основе `Task`, компилятор способен точно знать, как долго все остается в области видимости. Это позволяет ему генерировать обратные вызовы за вас, и в рамках этого он может организовать своевременный вызов оператором `using` метода `Dispose`.

В коде, критичном к производительности, вам может понадобиться учитывать затраты на анонимные функции. Если анонимная функция использует переменные из внешней области видимости, то в дополнение к объекту делегата, который вы создаете для ссылки на анонимную функцию, вы можете создать еще один: экземпляр класса, созданного для хранения общих локальных переменных. Компилятор будет по возможности повторно использовать эти хранители переменных — например, если один метод содержит две анонимные функции, они могут иметь возможность использовать объект совместно. Даже при такой оптимизации вы по-прежнему создаете дополнительные объекты, увеличивая нагрузку на сборщик мусора. (И в некоторых случаях вы можете в конечном итоге создавать этот объект, даже если никогда не попадете в ветку кода, которая создает делегат.) Это не особенно дорого — обычно это небольшие объекты, — но если вы сталкиваетесь с особенно удручающей проблемой производительности, вы можете попробовать улучшить ситуацию с помощью более многословного кода, который уменьшит количество объектов выделения.



Локальные функции не всегда влекут за собой такие же издержки. Когда локальная функция использует переменные своего внешнего метода, она не продлевает их время жизни. Поэтому компилятору не нужно создавать объект в куче для хранения общих переменных. Он по-прежнему создает тип для хранения всех общих переменных, но определяет его как структуру, которую передает по ссылке в качестве скрытого аргумента, не имея необходимости в блоке кучи. (Если вы создаете делегат, который ссылается на локальную функцию, он не может использовать эту оптимизацию и возвращается к той же стратегии, что используется в анонимных функциях, помещая общие переменные в объект в куче.)

Менее очевидно то, что использование локальных переменных из внешней области видимости в анонимной функции продлевает их время жизни, а это может означать, что сборщик мусора будет тратить больше времени на обнаружение того, что объекты, на которые ссылаются эти переменные, больше не используются. Как вы, возможно, помните из главы 7, CLR анализирует ваш код, чтобы определить, используются ли переменные. Это дает возможность освобождать объекты, не дожидаясь, пока ссылающиеся на них переменные выйдут из области видимости. Это позволяет значительно раньше очищать память, используемую некоторыми объектами, особенно в методах, выполнение которых занимает много времени. Но такой анализ

применим только к обычным локальным переменным. Его нельзя применять к переменным, которые используются в анонимной функции, потому что компилятор преобразует эти переменные в поля. (С точки зрения CLR они вообще не являются локальными переменными.) Поскольку C# обычно помещает все эти преобразованные переменные для конкретной области видимости в один объект, то ни один из объектов, на которые ссылаются эти переменные, не может быть освобожден до завершения метода, когда объект, содержащий переменные, сам становится недоступным. Это может означать, что в некоторых случаях можно получить ощутимый выигрыш от присвоения локальной переменной значения `null` после окончания работы с ней, позволяя освободить память этого конкретного объекта при следующей сборке мусора. (В обычных обстоятельствах это плохой совет и даже с анонимными функциями может не иметь полезного эффекта. Это следует делать только в том случае, если тестирование производительности показывает явное преимущество. Но к этому стоит обращаться в тех случаях, когда вы сталкиваетесь с проблемами производительности, связанными со сборкой мусора, и интенсивно используете продолжительные анонимные функции.)

Перехват переменных также может иногда приводить к ошибкам, особенно из-за связанной с областью видимости тонкости, присущей циклам `for`. (Раньше это затрагивало и циклы `foreach`, но Microsoft изменила поведение `foreach` в C# 5.0, решив, что исходное поведение — это не то, чего на самом деле хотят разработчики.) Листинг 9.35 сталкивается с этой проблемой.

Листинг 9.35. Проблемный захват переменных в цикле `for`

```
public static void Caught()
{
    var greaterThanN = new Predicate<int>[10];
    for (int i = 0; i < greaterThanN.Length; ++i)
    {
        greaterThanN[i] = value => value > i; // Bad use of i
    }

    Console.WriteLine(greaterThanN[5](20));
    Console.WriteLine(greaterThanN[5](6));
}
```

Этот пример инициализирует массив из делегатов `Predicate<int>`, где каждый делегат проверяет, больше ли значение определенного числа. (Кстати, вам не обязательно использовать массивы, чтобы увидеть проблему, которую я собираюсь описать. Вместо этого ваш цикл может передать создаваемые

делегаты в один из описываемых в главе 16 механизмов, которые обеспечивают параллельную обработку с выполнением кода в нескольких потоках. Но массивы облегчают выявление проблемы.) В частности, он сравнивает значение с `i`, счетчиком цикла, который определяет, в какую ячейку массива отправляется каждый делегат. Следовательно, элемент с индексом 5 ссылается на метод, который сравнивает свой аргумент с числом 5. Если бы это было так, код показал бы `True` дважды. Но факт в том, что он отображает `True`, а затем `False`. Оказывается, что листинг 9.35 создает массив делегатов, где каждый отдельный элемент сравнивает свой аргумент с числом 10.

Тем, кто сталкивается с этим впервые, это обычно кажется удивительным. Хорошенько подумав, достаточно легко понять, почему это происходит, особенно когда вы знаете, каким образом компилятор C# позволяет анонимной функции использовать переменные из содержащей ее области видимости. Цикл `for` объявляет переменную `i`, и, поскольку она используется не только содержащим методом `Caught`, но и каждым создаваемым в цикле делегатом, компилятор создает класс, аналогичный классу в листинге 9.32, и переменная помещается в поле этого класса. Поскольку переменная входит в область видимости при запуске цикла и остается там в течение всего цикла, компилятор создаст один экземпляр генерируемого им класса, и он будет использоваться всеми делегатами. Таким образом, когда цикл увеличивает `i`, это меняет поведение всех делегатов, поскольку все они используют одну и ту же переменную `i`.

По сути, проблема в том, что есть только одна переменная `i`. Вы можете исправить код, добавив новую переменную внутри цикла. В листинге 9.36 значение `i` копируется в другую локальную переменную, `current`, которая не входит в область видимости, пока не начнется итерация, и выходит из области видимости в конце каждой итерации. Итак, хотя существует только одна переменная `i`, которая действительна до тех пор, пока выполняется цикл, мы каждый проход цикла пользуемся новой переменной `current`. Поскольку каждый делегат получает собственную отдельную переменную `current`, каждый делегат в массиве сравнивает свой аргумент со значением, которое счетчик цикла имел во время этой конкретной итерации.

Компилятор по-прежнему генерирует класс, аналогичный классу в листинге 9.32, предназначенный для хранения переменной `current`, которая является общей для встроенных и содержащих методов. Только теперь он каждый раз создает новый экземпляр этого класса в цикле, чтобы дать каждой анонимной функции собственный экземпляр этой переменной. (Это

происходит автоматически, когда вы используете цикл `foreach`, потому что его правила области видимости немного отличаются: область видимости его итерационной переменной указана для каждой итерации, так что при каждой итерации это логически другой экземпляр переменной, поэтому нет необходимости добавлять в тело цикла дополнительную переменную, как мы это сделали в случае с `for`.)

Листинг 9.36. Модификация цикла для захвата текущего значения

```
for (int i = 0; i < greaterThanN.Length; ++i)
{
    int current = i;
    greaterThanN[i] = value => value > current;
}
```

Вы можете спросить, что произойдет, если написать анонимную функцию, которая использует переменные в нескольких областях видимости. Листинг 9.37 объявляет переменную с именем `offset` перед циклом, а лямбда-выражение использует и ее, и переменную, область видимости которой распространяется только на одну итерацию.

Листинг 9.37. Захват переменных в разных областях видимости

```
int offset = 10;
for (int i = 0; i < greaterThanN.Length; ++i)
{
    int current = i;
    greaterThanN[i] = value => value > (current + offset);
}
```

В этом случае компилятор создаст два класса — один для хранения любых общих переменных для каждой итерации (в данном случае `current`) и один для тех, чья область видимости охватывает весь цикл (в данном случае `offset`). Целевым объектом каждого делегата будет объект, содержащий переменные внутренней области видимости и ссылку на внешнюю область видимости.

На рис. 9.1 показано, как это будет работать, хотя он упрощен и показывает только первые пять элементов. Переменная `greatThanN` содержит ссылку на массив. Каждый элемент массива содержит ссылку на делегата. Каждый делегат ссылается на один и тот же метод, но у каждого из них есть свой целевой объект, которым каждый делегат объект может захватывать разные экземпляры переменной `current`. Каждый из этих целевых

объектов ссылается на единственный объект, содержащий переменную `offset`, захваченную из области видимости за пределами цикла.

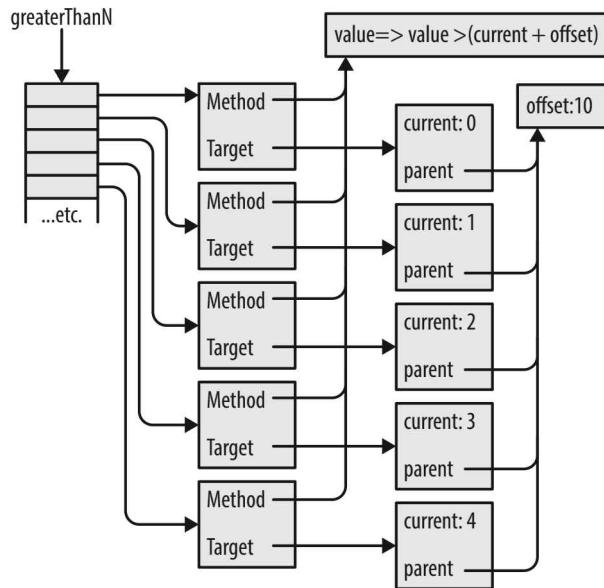


Рис. 9.1. Делегаты и захваченные области видимости

Лямбда-выражения и деревья выражений

У лямбды есть еще один козырь в рукаве помимо предоставления делегатов. Некоторые лямбды создают структуру данных, которая представляет собой код. Это происходит, когда вы используете лямбда-синтаксис в контексте, который требует `Expression<T>`, где `T` — тип делегата. Сам по себе `Expression<T>` не является типом делегата; это специальный тип в библиотеке классов .NET (в пространстве имен `System.Linq.Expressions`), который запускает в компиляторе альтернативную обработку лямбда-выражений. Листинг 9.38 использует этот тип.

Листинг 9.38. Лямбда-выражение

```
Expression<Func<int, bool>> greaterThanZero = value => value > 0;
```

Этот пример напоминает некоторые лямбды и делегаты, которые я уже показывал в этой главе, но компилятор обрабатывает его совершенно по-другому. Он не будет генерировать метод, т. е. не будет скомпилированного

промежуточного языка, представляющего собой тело лямбды. Вместо этого компилятор выдаст код, подобный коду в листинге 9.39.

Листинг 9.39. Что компилятор делает с лямбда-выражением

```
ParameterExpression valueParam = Expression.Parameter(typeof(int),  
"value");  
ConstantExpression constantZero = Expression.Constant(0);  
BinaryExpression comparison = Expression.GreaterThan(valueParam,  
constantZero);  
Expression<Func<int, bool>> greaterThanZero =  
    Expression.Lambda<Func<int, bool>>(comparison, valueParam);
```

Для создания объекта для каждого подвыражения в лямбда-выражении этот код вызывает различные фабричные функции, предоставляемые классом `Expression`. Все начинается с простых операндов — параметра `value` и константного значения 0. Они передаются в объект, представляющий собой сравнение «больше, чем», которое, в свою очередь, становится телом объекта, представляющего собой все лямбда-выражение.

Возможность создания объектной модели для выражения позволяет написать API, в котором поведение контролируется структурой и содержанием выражения. Например, некоторые API доступа к данным могут принимать выражение, похожее на показанное в листингах 9.38 и 9.39, и использовать его для создания части запроса к базе данных. Я буду говорить об интегрированных запросах C# в главе 10, но листинг 9.40 дает представление о том, как лямбда-выражение можно использовать в качестве основы для запроса.

Листинг 9.40. Выражения и запросы к базе данных

```
var expensiveProducts = dbContext.Products.Where(p => p.ListPrice > 3000);
```

В этом примере используется библиотека Microsoft, называемая `Entity Framework`, но другие технологии доступа к данным тоже могут использовать такой подход. В этом примере метод `Where` принимает аргумент типа `Expression<Func<Product, bool>>`. `Product` — это класс, который соответствует сущности в базе данных, но важным моментом здесь является использование `Expression<T>`⁵. Оно означает, что компилятор будет генерировать код,

⁵ Вы можете удивиться, встретив здесь `Func<Product, bool>` вместо `Predicate<Product>`. Метод `Where` является частью функционала .NET под названием LINQ, которая широко использует делегаты. Чтобы избежать определения огромного количества новых типов делегатов, LINQ использует типы `Func`, а для согласованности по всему API предпочитает `Func`, даже если подходят другие стандартные типы.

который создает дерево объектов со структурой, соответствующей этому лямбда-выражению. Метод `Where` обрабатывает это дерево выражений, генерируя SQL-запрос, включающий следующий пункт: `WHERE [Extent1].[ListPrice] > cast(3000 as decimal(18))`.

Таким образом, хотя я и написал свой запрос в виде выражения C#, работа по поиску подходящих объектов будет выполняться на моем сервере базы данных.

Лямбда-выражения были добавлены в C# для обеспечения такого рода обработки запросов в составе набора функций, известных под общим названием LINQ (о чем рассказывает глава 10). Однако, как и в случае с большинством функций, относящихся к LINQ, их можно использовать для других целей. Например, они использованы в популярной библиотеке .NET, применяемой в автоматизированном тестировании под названием Moq (<https://github.com/moq>). С целью тестирования Moq создает поддельные реализации интерфейсов и использует лямбда-выражения, чтобы предоставить простой API для настройки поведения этих подделок. Листинг 9.41 использует класс `Mock<T>` из Moq для создания поддельной реализации интерфейса .NET `IEqualityComparer<string>`. Код вызывает метод `Setup`, который принимает выражение, указывающее на конкретный вызов, для которого мы хотели бы определить специальную обработку. В нашем случае, если фальшивая реализация `IEqualityComparer<string>.Equals` вызывается с аргументами «`Color`» и «`Colour`», мы хотим, чтобы она возвращала `true`.

Листинг 9.41. Использование лямбда-выражений библиотекой Moq

```
var fakeComparer = new Mock<IEqualityComparer<string>>();
fakeComparer
    .Setup(c => c.Equals("Color", "Colour"))
    .Returns(true);
```

Если бы этот аргумент `Setup` был обычным делегатом, у Moq не было бы возможности его проверить. Но поскольку это дерево выражений, Moq может проанализировать его и найти то, о чем мы просим.



К сожалению, деревья выражений – это область C#, которая отстает от остальной части языка. Они были введены еще в C# 3.0, но различные языковые функции, такие как поддержка кортежей и асинхронных выражений, все еще не могут использоваться в дереве выражений, потому что объектная модель не имеет способа их представления.

События

Иногда полезно, чтобы объекты могли отправлять уведомления о том, что что-то произошло. Например, через каркас создания пользовательского интерфейса на стороне клиента вам может понадобиться узнать, что пользователь нажал одну из кнопок вашего приложения. Делегаты предоставляют базовый механизм обратного вызова, необходимый для уведомлений, но есть множество способов их использования. Должен ли делегат передаваться как аргумент метода, аргумент конструктора или, возможно, как свойство? Как вы должны осуществлять отписку от уведомлений? CTS формализует ответы на эти вопросы с помощью специальной разновидности члена класса под названием «событие», а C# предоставляет синтаксис для работы с событиями. Листинг 9.42 показывает класс с одним членом события.

Листинг 9.42. Класс с событием

```
public class Eventful
{
    public event Action<string> Announcement;

    public void Announce(string message)
    {
        Announcement?.Invoke(message);
    }
}
```

Как и с другими членами, вы можете начать с указания видимости, которая по умолчанию будет установлена в `private`. Затем следует ключевое слово `event`, указывающее, что речь идет о событии. После него указывается тип события, который может быть любым типом делегата. Я использовал `Action<string>`, хотя, как вы скоро увидите, это не самый стандартный выбор. Наконец, мы указываем имя члена, так что этот пример определяет событие с именем `Announcement`.

Чтобы обработать событие, вы должны предоставить делегат нужного типа, а также использовать синтаксис `+=`, чтобы подключить этот делегат в качестве обработчика. В листинге 9.43 используется лямбда, но вы можете использовать любое выражение, которое создает или неявно преобразуется в делегат того типа, который требуется для события.

В листинге 9.42 также показано, как вызвать событие, т. е. все обработчики, которые были подключены к событию. В его методе `Announce` ис-

пользован тот же синтаксис, который мы бы использовали в случае, если бы `Announcement` было полем, содержащим вызываемый делегат. По факту, если говорить о коде внутри класса, именно так и выглядит событие — как поле. Я решил использовать член делегата `Invoke` явно вместо того, чтобы писать `Announcement (message)` (несмотря на то, что ранее я заявлял, что обычно предпочитаю другой подход), потому что это позволило мне применить `null`-условный оператор (`? .`). Компилятор создаст код, который вызывает делегат, только если тот не равен `null`. В противном случае мне пришлось бы использовать оператор `if` для проверки того, что в момент вызова поле не содержит `null`.

Листинг 9.43. Обработка событий

```
var source = new Eventful();
source.Announcement += m => Console.WriteLine("Announcement: " + m);
```

Так зачем нужен специальный тип члена, если он выглядит как поле? Ну, на поле он похож только внутри определяющего класса. Код за пределами класса не может вызвать событие, поэтому листинг 9.44 не будет компилироваться.

Листинг 9.44. Как нельзя вызывать событие

```
var source = new Eventful();
source.Announcement("Will this work?"); // Нет, это даже не скомпилируется
```

Единственное, что можно сделать с событием извне, это присоединить к нему обработчик с помощью `+=` или удалить его с помощью `-=`. Синтаксис для добавления и удаления обработчиков событий необычен в том смысле, что это единственный случай в C#, когда `+=` и `-=` можно использовать без соответствующих автономных операторов `+` или `-`. Действия, выполняемые `+=` и `-=` по отношению к событиям, — это скрытые вызовы методов. Как и свойства, события — это в действительности пары методов с особым синтаксисом. По своей идее они похожи на код, показанный в листинге 9.45. (На самом деле реальный код включает в себя кое-какой умеренно сложный неблокируемый, потокобезопасный код. Я убрал его, потому что многопоточность может скрыть основную идею.) Однако эффект будет другим, потому что ключевое слово `event` добавляет к типу метаданные, помечая методы как события, так что эта ремарка сделана для иллюстрации.

Как и в случае со свойствами, события существуют главным образом для предоставления удобного отличительного синтаксиса, а также для того,

чтобы инструментам было проще представлять предлагаемый классами функционал. События особенно важны при работе с элементами пользовательского интерфейса. В большинстве каркасов для разработки пользовательского интерфейса объекты, представляющие интерактивные элементы, часто способны вызывать широкий диапазон событий, соответствующих различным формам ввода, таким как клавиатура, мышь или касание. Также часто встречаются события, относящиеся к поведению, характерному для конкретного элемента управления, например выбор нового элемента в списке. Поскольку CTS определяет стандартную идиому, с помощью которой элементы могут инициировать события, визуальные средства разработки пользовательского интерфейса, включая встроенные в Visual Studio, способны отображать доступные события и генерировать за вас обработчики.

Листинг 9.45. Примерный эффект объявления события

```
private Action<string> Announcement;

// Не настоящий код.
// Реальный код более сложный и поддерживает параллельные вызовы.
public void add_Announcement(Action<string> handler)
{
    Announcement += handler;
}
public void remove_Announcement(Action<string> handler)
{
    Announcement -= handler;
}
```

Стандартный шаблон делегата события

Событие в листинге 9.42 примечательно тем, что в нем используется тип делегата `Action<T>`. Это допустимо, но на практике встречается редко, потому что почти во всех событиях используются типы делегатов, соответствующие определенному шаблону. Этот шаблон требует, чтобы сигнатура метода делегата содержала два аргумента. Тип первого аргумента — `object`, а второго — либо `EventArgs`, либо тип, производный от `EventArgs`. В листинге 9.46 показан тип делегата `EventHandler` в пространстве имен `System`, который является самым простым и наиболее широко используемым примером этого шаблона.

Первый аргумент обычно называется `sender`, потому что через этот аргумент источник события передает ссылку на самого себя. Это означает, что, если

вы присоединяете один делегат к нескольким источникам событий, обработчик всегда будет знать, какой именно источник отправил конкретное уведомление.

Листинг 9.46. Тип делегата EventHandler

```
public delegate void EventHandler(object sender, EventArgs e);
```

Второй аргумент предоставляет место для размещения информации, относящейся к событию. Например, элементы пользовательского интерфейса WPF определяют различные события для обработки ввода с помощью мыши, которые используют более специализированные типы делегатов, например `MouseButtonEventHandler`. Их сигнатуры включают в себя соответствующий специализированный аргумент события, который содержит подробные сведения о событии. Например, `MouseButtonEventArgs` определяет метод `GetPosition`, который сообщает, где находилась мышь при нажатии кнопки, и определяет различные другие свойства, хранящие дополнительную информацию, включая `ClickCount` и `Timestamp`.

Каким бы специализированным ни был тип второго аргумента, он всегда будет производным от базового типа `EventArgs`. Этот базовый тип не очень интересен — он не добавляет членов помимо стандартных, определяемых типом `object`. Тем не менее это позволяет написать метод общего назначения, который можно прикрепить к любому событию, использующему этот шаблон. Правила совместимости делегатов означают, что даже если тип делегата указывает в качестве типа второго аргумента `MouseButtonEventArgs`, метод, второй аргумент которого имеет тип `EventArgs`, остается допустимой целью. Иногда это может быть полезно для генерации кода или других инфраструктурных сценариев. Однако главное преимущество стандартного шаблона событий — это осведомленность опытных разработчиков C# относительно того, что события работают именно так.

Пользовательские методы добавления и удаления

Иногда вам может понадобиться отказаться от реализации событий по умолчанию, созданной компилятором C#. Например, класс может определять слишком много событий, большинство из которых не будут использоваться в подавляющем числе случаев. Это часто характеризует каркасы для разработки пользовательского интерфейса. Пользовательский интерфейс WPF может иметь тысячи элементов, каждый из которых предлагает более

100 событий, но обычно вы подключаете обработчики только к нескольким из них, но даже в этом случае вы обрабатываете только часть имеющихся событий. В таком случае выделять поле для каждого доступного события каждого элемента будет неэффективно.

Использование реализации на основе полей по умолчанию для большого количества редко используемых событий может добавить сотни байтов к занимаемому каждым элементом пользовательского интерфейса пространству, а это способно оказать заметное влияние на производительность. (В случае WPF это может составить несколько сотен тысяч байтов. Это может показаться не таким уж большим объемом, учитывая возможности памяти современных компьютеров, но это способно поставить ваш код в ситуацию, в которой он больше не сможет эффективно использовать кэш ЦП, а это приведет к резкому снижению скорости отклика приложений. Даже если размер кэша составляет несколько мегабайтов, самые быстрые его части обычно намного меньше, так что потеря нескольких сотен килобайтов в критической структуре данных может существенно повлиять на производительность.)

Другая причина, по которой вы можете отказаться от реализации событий, создаваемых компилятором по умолчанию, заключается в том, что вам может потребоваться более сложная семантика вызова событий. Например, WPF поддерживает *всплытие событий*: если элемент пользовательского интерфейса не обрабатывает определенные события, они будут переданы родительскому элементу, затем его родительскому элементу и далее вверх по дереву, пока не будет найден обработчик или достигнута вершина. Хотя можно было бы реализовать такую схему с помощью стандартных средств реализации событий C#, в случае, когда обработчики событий относительно редки, возможны гораздо более эффективные стратегии.

Для поддержки этих сценариев C# дает вам возможность определять собственные методы добавления и удаления для события. Снаружи ваше событие будет выглядеть обычным — любой, кто использует ваш класс, будет использовать тот же синтаксис `+=` и `-=` для добавления и удаления обработчиков, и будет невозможно сказать, что событие содержит пользовательскую реализацию. Листинг 9.47 показывает класс с двумя событиями, и он использует единый словарь для всех экземпляров класса, чтобы отслеживать, какие события для каких объектов были обработаны. Подход расширяем для большего числа событий — словарь использует пары объектов в качестве ключа, поэтому каждая запись представляет определенную пару (источник, событие). (Кстати, этот код не финального качества. Он

небезопасен в случае многопоточного использования, а также ведет к утечке памяти, если экземпляр `ScarceEventSource`, к которому все еще подключены обработчики событий, выходит из использования. Пример просто показывает, как выглядят пользовательские обработчики событий; это не полностью готовое решение.)

Листинг 9.47. Пользовательские `add` и `remove` для немногочисленных событий

```
public class ScarceEventSource
{
    // Один словарь, общий для всех экземпляров этого класса,
    // Отслеживание всех обработчиков для всех событий.
    // Остерегайтесь утечек памяти – этот код только для иллюстрации.
    private static readonly
        Dictionary<(ScarceEventSource, object),
        EventHandler> _eventHandlers = new Dictionary<(ScarceEventSource,
        object), EventHandler>();

    // Объекты, используемые в качестве ключей для идентификации
    // определенных событий в словаре.
    private static readonly object EventOneId = new object();
    private static readonly object EventTwoId = new object();

    public event EventHandler EventOne
    {
        add
        {
            AddEvent(EventOneId, value);
        }
        remove
        {
            RemoveEvent(EventOneId, value);
        }
    }

    public event EventHandler EventTwo
    {
        add
        {
            AddEvent(EventTwoId, value);
        }
        remove
        {
            RemoveEvent(EventTwoId, value);
        }
    }
}
```

```
public void RaiseBoth()
{
    RaiseEvent(EventOneId, EventArgs.Empty);
    RaiseEvent(EventTwoId, EventArgs.Empty);
}

private (ScarceEventSource, object) MakeKey(object eventId)
        => (this, eventId);

private void AddEvent(object eventId, EventHandler handler)
{
    var key = MakeKey(eventId);
    eventHandlers.TryGetValue(key, out EventHandler entry);
    entry += handler;
    _eventHandlers[key] = entry;
}

private void RemoveEvent(object eventId, EventHandler handler)
{
    var key = MakeKey(eventId);
    EventHandler entry = _eventHandlers[key];
    entry -= handler;
    if (entry == null)
    {
        _eventHandlers.Remove(key);
    }
    else
    {
        _eventHandlers[key] = entry;
    }
}

private void RaiseEvent(object eventId, EventArgs e)
{
    var key = MakeKey(eventId);
    if (_eventHandlers.TryGetValue(key, out EventHandler handler))
    {
        handler(this, e);
    }
}
```

Синтаксис пользовательских событий напоминает полный синтаксис свойства: мы добавляем блок после объявления, содержащего два члена, хотя они называются `add` и `remove` вместо `get` и `set`. (В отличие от свойств, вы всегда должны предоставлять оба метода.) Это отключает генерацию поля,

которое обычно содержит событие, что означает, что класс `ScarceEventSource` не имеет полей экземпляров вообще — экземпляры этого типа настолько малы, насколько это возможно для объекта.

Цена за этот небольшой объем памяти — значительное увеличение сложности; я написал примерно в 16 раз больше строк кода, чем нужно для событий, создаваемых компилятором, и понадобится еще больше, чтобы исправить описанные ранее недостатки. Более того, этот метод обеспечивает преимущество, только если события действительно обрабатываются редко, — если бы я прикреплял обработчики к обоим событиям каждого экземпляра этого класса, хранилище на основе словаря потребляло бы больше памяти, чем просто поля для каждого события в каждом экземпляре класса. Таким образом, вы должны использовать этот тип пользовательской обработки событий только в случае, если вам требуется нестандартное поведение вызова событий или если вы твердо уверены, что действительно сэкономите память и что экономия того стоит.

События и сборщик мусора

Что касается сборки мусора, то делегаты являются обычными объектами. Если сборщик мусора обнаруживает, что экземпляр делегата достижим, он проверяет свойство `Target`, и любой объект, на который оно ссылается, тоже будет считаться достижимым наряду с любыми объектами, на которые он, в свою очередь, ссылается. Хотя в этом нет ничего примечательного, могут возникнуть ситуации, когда оставление подключенных обработчиков событий может привести к зависанию объектов в памяти, тогда как вы ожидаете, что они будут собраны сборщиком мусора.

Делегатам и событиям не присуще ничего, что может повысить вероятность нарушить работу сборщика мусора. Если вы получите утечку памяти, связанную с событием, она будет иметь ту же структуру, что и любая другая утечка памяти .NET: начиная с корневой ссылки, будет существовать цепочка ссылок, которая сделает объект достижимым даже после того, как вы закончили его использовать. Единственная причина, по которой события получают особую вину за утечки памяти, заключается в том, что они часто используются способами, которые могут вызвать проблемы.

Например, предположим, что ваше приложение поддерживает некоторую объектную модель, представляющую его состояние, а ваш код пользователь-

ского интерфейса находится на отдельном уровне, который использует эту базовую модель, адаптируя содержащуюся в нем информацию для показа на экране. Такое разбиение на уровни правильно, потому что смешивать код, который взаимодействует с пользователем, и код, который реализует логику приложения, — плохая идея. Но проблема возникает, если базовая модель объявляет об изменениях в состоянии, которые должен отразить пользовательский интерфейс. Если эти изменения объявляются через события, ваш код пользователяского интерфейса, как правило, прикрепляется к ним обработчики.



Существует устойчивый миф о том, что подобная утечка памяти на основе событий имеет отношение к циклическим ссылкам. На самом деле сборщик мусора отлично справляется с циклическими ссылками. Это правда, что в таких сценариях часто есть циклические ссылки, но дело здесь не в этом. Проблема кроется в случайном сохранении доступности объектов после того, как они вам больше не нужны. Это приведет к проблемам независимо от наличия циклических ссылок.

Теперь представьте, что кто-то закрывает одно из окон вашего приложения. Можно надеяться, что объекты, представляющие пользовательский интерфейс этого окна, будут зарегистрированы как недостижимые при следующем запуске сборщика мусора. Библиотека пользовательского интерфейса, скорее всего, постарается, чтобы так оно и было. Например, WPF обеспечивает доступность каждого экземпляра своего класса `Window` до тех пор, пока соответствующее окно открыто, но, как только окно закрыто, она прекращает удерживать ссылки на окно, чтобы позволить очистить все объекты пользовательского интерфейса этого окна.

Однако если вы обрабатываете событие из модели вашего основного приложения с помощью метода в классе, производном от `Window`, и явно не удаляете этот обработчик при закрытии окна, возникает проблема. Пока ваше приложение работает, что-то где-то, скорее всего, будет обеспечивать достижимость базовой модели вашего приложения. Это означает, что целевые объекты любых делегатов, удерживаемых вашей моделью приложения (например, делегатов, которые были добавлены в качестве обработчиков событий), будут по-прежнему достижимы, что не позволит сборщику мусора их удалить. Таким образом, если производный от `Window` объект для теперь уже закрытого окна по-прежнему обрабатывает события из модели приложения, это окно — и все содержащиеся в нем элементы пользователь-

ского интерфейса — по-прежнему будет доступно и не будет собираться сборщиком.

Вы можете разобраться с этим, убедившись, что если ваш уровень пользовательского интерфейса присоединяет обработчики к объектам, которые будут оставаться действительными в течение длительного времени, вы удаляете эти обработчики, когда соответствующий элемент пользовательского интерфейса больше не используется. В качестве альтернативы вы можете использовать слабые ссылки, чтобы гарантировать, что если ваш источник событий является единственной сущностью, содержащей ссылку на цель, он не будет ее удерживать. WPF может помочь вам в этом — он предоставляет класс `WeakEventManager`, который позволяет вам обрабатывать событие так, чтобы обрабатываемый объект мог быть собран без необходимости отписываться от события. WPF сам использует этот метод при привязке интерфейса пользователя к источнику данных, который предоставляет события уведомлений об изменении свойств.



Хотя в пользовательском интерфейсе часто возникают утечки, связанные с событиями, они могут возникать где угодно. Пока источник события остается достижимым, все его подключенные обработчики тоже будут достижимы.

Сравнение событий и делегатов

Некоторые API предоставляют уведомления через события, в то время как другие напрямую используют делегаты. Как же решить, какой подход использовать? В некоторых случаях решение может быть принято за вас, потому что вы хотите соответствовать какой-то конкретной идиоме. Например, если вы хотите, чтобы ваш API поддерживал новый асинхронный функционал C#, вам придется реализовать шаблон, описанный в главе 17, который использует для обратных вызовов завершения делегаты, а не события. События, с другой стороны, предоставляют понятный способ подписки и отписки, что делает их наилучшим выбором в некоторых ситуациях. Соглашение — это еще одна причина: если вы пишете элемент пользовательского интерфейса, события, скорее всего, будут подходящим решением, потому что это преобладающая идиома.

В случаях, когда ограничения или соглашения не дают ответа, вам придется подумать о том, как будет использоваться обратный вызов. Если на уведом-

ление будет несколько подписчиков, событие может оказаться лучшим выбором. Это не является абсолютно необходимым, потому что любой делегат может быть многоадресным, но по соглашению такое поведение обычно реализуется через события. Если пользователям вашего класса в какой-то момент понадобится удалить обработчик, события также будут хорошим выбором. При этом интерфейс `Iobservable` также поддерживает отписку и может быть хорошим выбором, если вам нужны более продвинутые функциональные возможности. Этот интерфейс является частью `Reactive Extensions` для .NET и описан в главе 11.

Если разумно иметь только один целевой метод, то обычно вы передаете делегат в качестве аргумента методу или конструктору. Например, если тип делегата имеет отличное от `void` возвращаемое значение, от которого зависит API (например, `bool`, возвращаемый предикатом, переданным в `Array.FindAll`), бессмысленно иметь несколько целей или нулевые цели. В данном случае событие — неподходящая идиома, потому что его модель, ориентированная на подписку, считает совершенно нормальным присоединять несколько обработчиков или не присоединять их вообще.

Иногда разумно иметь либо нулевой обработчик, либо не более одного обработчика. Например, возьмем класс `CollectionView` из WPF, который может сортировать, группировать и фильтровать данные из коллекции. Вы настраиваете фильтрацию, передавая `Predicate<object>`. Он не передается в качестве аргумента конструктора, поскольку фильтрация является необязательной, поэтому вместо этого класс определяет свойство `Filter`. Событие будет здесь неуместным отчасти потому, что `Predicate<object>` не соответствует обычному шаблону делегата события, но главным образом потому, что классу нужен однозначный ответ «да» или «нет», так что он не желает поддерживать несколько целей. (Тот факт, что все типы делегатов поддерживают многоадресность, означает, что предоставление нескольких целей остается возможным. Но решение использовать свойство, а не событие сигнализирует о том, что пытаться предоставлять несколько обратных вызовов в данном случае бесполезно.)

Сравнение делегатов и интерфейсов

В начале этой главы я утверждал, что делегаты предлагают менее громоздкий механизм для обратных вызовов и уведомлений, нежели интерфейсы. Так почему же некоторые API требуют от вызывающей стороны для включения

обратных вызовов реализации интерфейса? Почему у нас есть `IComparer<T>`, а не делегат? На самом деле у нас имеется и то и другое — есть тип делегата под названием `Comparision<T>`, который поддерживается многими API в качестве альтернативы `IComparer<T>`. Массивы и `List<T>` содержат перегрузки своих методов `Sort`, которые принимают оба варианта.

В ряде ситуаций объектно ориентированный подход может оказаться предпочтительнее использования делегатов. Объект, который реализует `IComparer<T>`, может предоставить свойства для настройки способа сравнения (например, возможность выбора между различными критериями сортировки). Возможно, вы захотите собрать и суммировать информацию по нескольким обратным вызовам, и, хотя это можно делать с помощью захваченных переменных, может оказаться удобнее вернуть информацию в конце, если она доступна через свойства объекта.

Это хорошее решение для тех, кто пишет код, который является целью обратных вызовов, а не для разработчика кода, который этот вызов осуществляет. В конечном итоге делегаты выглядят более гибкими, поскольку позволяют пользователю API решать, как структурировать свой код, тогда как интерфейс накладывает свои ограничения. Однако если интерфейс совпадает с нужными абстракциями, делегаты могут показаться раздражающей лишней деталью. Вот почему некоторые API, такие как API сортировки, позволяют использовать и вариант, который принимает `IComparer<T>`, и вариант, который принимает `Comparision<T>`.

Есть ситуация, в которой интерфейсы могут быть предпочтительнее делегатов. Это происходит тогда, когда вам нужно предоставить несколько взаимосвязанных обратных вызовов. `Reactive Extensions` для .NET определяет абстракцию для уведомлений, которая дает возможность узнать, когда вы достигли конца последовательности событий или когда произошла ошибка. Поэтому в этой модели подписчики реализуют интерфейс с тремя методами: `OnNext`, `OnCompleted` и `OnError`. Использование интерфейса имеет смысл, потому что для полной подписки обычно требуются все три метода.

Итог

Делегаты — это объекты, предоставляющие ссылку на метод, который может быть статическим методом или методом экземпляра. При использовании методов экземпляра делегат также содержит ссылку на целевой объект,

поэтому коду, который вызывает делегат, не нужно ее указывать. Делегаты могут ссылаться на несколько методов, хотя это усложняет ситуацию в случае, когда тип возвращаемого значения делегата отличается от `void`. Хотя CLR обрабатывает типы делегатов особым образом, они по-прежнему остаются просто ссылочными типами, а это означает, что ссылку на делегат можно передать в качестве аргумента, вернуть из метода и сохранить в поле, переменной или свойстве. Тип делегата определяет сигнатуру целевого метода. Это реализовано с помощью метода `Invoke` типа, но C# может его скрыть, предлагая синтаксис, в котором вы можете вызывать выражение делегата напрямую, без явной ссылки на `Invoke`. Вы можете создать делегат, который ссылается на любой метод с совместимой сигнатурой. Вы также можете заставить C# выполнять большую часть работы за вас. Если вы пишете анонимную функцию, C# предоставит вам подходящее объявление, а также может выполнить закулисную работу, чтобы сделать переменные в содержащем методе доступными для внутреннего метода. Делегаты являются основой событий, которые обеспечивают формализованную модель публикации уведомлений и подписки на них.

Функционал C#, в котором особенно широко используются делегаты, — это LINQ, о котором пойдет речь в следующей главе.

ГЛАВА 10

LINQ

Язык интегрированных запросов (Language Integrated Query, LINQ) — это мощная коллекция языковых возможностей C# для работы с наборами информации. Он будет полезен в любом приложении, которое работает с множеством фрагментов данных (т. е. практически в любом приложении). Хотя одной из его первоначальных задач было обеспечение прямого доступа к реляционным базам данных, LINQ применим ко многим видам информации. Например, его можно использовать с объектными моделями в памяти, информационными службами на основе HTTP, JSON и документами XML.

LINQ — это не одна-единственная возможность. Он опирается на несколько языковых элементов, которые работают в связке. Наиболее видная языковая особенность, связанная с LINQ, — это *выражение запроса*, форма выражения, которая напоминает запрос к базе данных, но может использоваться для выполнения запросов к любому поддерживаемому источнику, включая старые добрые объекты. Как вы увидите, выражения запросов сильно зависят от ряда других языковых функций, таких как лямбда-выражения, методы расширения и объектные модели выражений.

Языковая поддержка — это не все. Для реализации набора запрашивающих примитивов, называемых *операторами LINQ*, нужны библиотеки классов. Каждый различный тип данных требует собственной реализации, и набор операторов для любого конкретного типа информации называется *провайдером LINQ*. (Кстати, их также можно использовать из Visual Basic и F#, поскольку эти языки также поддерживают LINQ.) Microsoft предоставляет несколько провайдеров, некоторые из которых встроены в библиотеку классов .NET, а некоторые доступны в виде отдельных пакетов NuGet. Например, существует провайдер для Entity Framework, системы объектно-реляционного отображения для работы с базами данных. Предлагается провайдер LINQ для облачной базы данных Cosmos DB (функция Microsoft Azure). А реактивные расширения для .NET (Rx), описанные в главе 11, обеспечивают поддержку LINQ для потоков данных в реальном времени.

Короче говоря, LINQ – это широко поддерживаемая и расширяемая идиома в .NET, поэтому вы легко найдете провайдеров с открытым исходным кодом и провайдеров от сторонних разработчиков.

В большинстве примеров в этой главе используется провайдер LINQ to Objects. Отчасти это связано с тем, что в данном случае не нужно загромождать примеры лишними подробностями вроде соединения с базой данных или службами. Однако есть и более важная причина. Появление LINQ в 2007 году значительно изменило мой стиль написания кода на C#, и это полностью заслуга LINQ to Objects. Хотя синтаксис запросов LINQ делает его похожим на технологию доступа к данным, по моему мнению, у него гораздо больше достоинств. Наличие сервисов LINQ для любой коллекции объектов делает его полезным в любой точке вашего кода.

Выражения запроса

Наиболее заметная особенность LINQ – это синтаксис выражения запроса. Это не самое главное – как мы увидим позже, вполне возможно продуктивно пользоваться LINQ без написания выражения запроса. Тем не менее это естественный синтаксис для многих видов запросов.

На первый взгляд выражение запроса чем-то напоминает запрос к базе данных, но синтаксис работает с любым провайдером LINQ. В листинге 10.1 показано выражение запроса, которое использует LINQ to Objects для поиска определенных объектов `CultureInfo`. (Объект `CultureInfo` содержит набор информации о конкретной культуре, такой как символ местной валюты, язык и т. д. В некоторых системах это называется *локалью*.) В этом конкретном запросе рассматривается символ, обозначающий десятичную точку. Многие страны используют запятую вместо точки, и в этих странах 100,000 означает число 100, записанное с точностью до трех знаков после запятой; в англоязычных культурах принята запись через точку 100.000. Выражение запроса просматривает все известные системы культуры и возвращает те, которые используют запятую в качестве десятичного разделителя.

Цикл `foreach` в этом примере показывает результаты запроса. В моей выводится список из 389 культур, что говорит о том, что чуть менее половины из 841 доступных культур используют запятую, а не десятичную точку. Конечно, я мог бы легко проделать это и без использования LINQ. Листинг 10.2 даст те же результаты.

Листинг 10.1. Выражение запроса LINQ

```
IEnumerable<CultureInfo> commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture;

foreach (CultureInfo culture in commaCultures)
{
    Console.WriteLine(culture.Name);
}
```

Листинг 10.2. Эквивалент без использования LINQ

```
CultureInfo[] allCultures = CultureInfo.GetCultures(
    CultureTypes.AllCultures);
foreach (CultureInfo culture in allCultures)
{
    if (culture.NumberFormat.NumberDecimalSeparator == ",")
    {
        Console.WriteLine(culture.Name);
    }
}
```

Оба примера имеют восемь непустых строк кода, хотя если игнорировать строки, содержащие лишь фигурные скобки, в листинге 10.2 окажется всего четыре строки, на две меньше, чем в листинге 10.1. Опять же, если мы посчитаем операторы, в примере с LINQ их будет только три, а не четыре, как в примере на основе цикла. Поэтому трудно убедительно показать, что один подход проще другого.

Однако листинг 10.1 имеет существенное преимущество: код, который решает, какие элементы выбрать, хорошо отделен от кода, который решает, что делать с этими элементами. Листинг 10.2 смешивает эти две задачи: код, который выбирает объекты, располагается наполовину снаружи и наполовину внутри цикла.

Другое отличие состоит в том, что листинг 10.1 имеет более декларативный стиль: он фокусируется на том, что мы хотим, а не на том, как этого добиться. Выражение запроса описывает элементы, которые нам нужны, не требуя, чтобы это было достигнуто каким-то конкретным способом. В интересах этого не очень полезного простого примера, как и в случае более сложных примеров, особенно использующих провайдера LINQ для доступа к базе данных, может оказаться крайне полезным дать провайдеру свободу в при-

нятии решения, как именно выполнять запрос. Подход листинга 10.2, при котором осуществляется перебор с помощью цикла `foreach` и выбор нужного элемента, был бы плохой идеей в случае взаимодействия с базой данных — обычно мы хотим, чтобы такую выборку выполнял сервер.

Запрос в листинге 10.1 состоит из трех частей. Все выражения запроса должны начинаться с выражения `from`, которое указывает на источник запроса. В нашем случае источником является массив типа `CultureInfo[]`, возвращаемый методом `GetCultures` класса `CultureInfo`. Помимо определения источника для запроса выражение `from` содержит имя, которым в данном случае является `culture`. Это называется переменной диапазона, и мы можем использовать ее в оставшейся части запроса для представления одного элемента из источника. Выражения могут выполняться много раз — выражение `where` в листинге 10.1 выполняется по разу для каждого элемента коллекции, поэтому переменная диапазона будет каждый раз иметь другое значение. Это напоминает переменную итерации в цикле `foreach`. Общая структура выражения `from` действительно похожа — у нас есть переменная, которая представляет элемент коллекции, ключевое слово `in` и источник, отдельные элементы которого будут представлять указанная переменная. Как переменная итерации цикла `foreach` находится в области видимости только внутри цикла, переменная диапазона `culture` действительно только внутри выражения запроса.



Хотя аналогии с `foreach` могут быть полезны для понимания сути запросов LINQ, не нужно воспринимать их слишком буквально. Например, не все провайдеры напрямую выполняют выражения в запросе. Некоторые провайдеры LINQ преобразуют выражения запросов в запросы к базе данных, и в этом случае код C# в различных выражениях внутри запроса не выполняется в традиционном смысле. Итак, хотя будет правильным сказать, что переменная диапазона представляет одно значение из источника, не всегда верно, что выражения будут выполняться один раз для каждого элемента, который они обрабатывают, причем значение диапазона будет содержать значение этого элемента. Это верно для листинга 10.1, потому что он использует LINQ to Objects, но это не обязательно так для всех провайдеров.

Вторая часть запроса в листинге 10.1 — это выражение `where`. Оно не является обязательным, но, если хотите, вы можете иметь несколько таких предложений в одном запросе. Выражение `where` фильтрует результаты,

а в этом примере утверждается, что мне нужны только объекты `CultureInfo` с `NumberFormat`, указывающим, что в качестве десятичного разделителя используется запятая.

Заключительная часть запроса — это выражение `select`, и все выражения запроса заканчиваются либо одним из таких выражений, либо выражением `group`. Это определяет окончательный результат запроса. Этот пример говорит, что мы хотим получить каждый объект `CultureInfo`, который не был исключен запросом. Цикл `foreach` в листинге 10.1, который выдает результаты запроса, использует только свойство `Name`, поэтому я мог бы написать запрос, который извлекал бы только его. Как показано в листинге 10.3, если я так поступлю, мне придется изменить цикл, потому что результирующий запрос теперь генерирует строки вместо объектов `CultureInfo`.

Листинг 10.3. Извлечение только одного свойства в запросе

```
IEnumerable<string> commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture.Name;

foreach (string cultureName in commaCultures)
{
    Console.WriteLine(cultureName);
}
```

Возникает вопрос: а какой тип имеют выражения запроса в целом? В листинге 10.1 `commaCultures` — это `IEnumerable<CultureInfo>`; в листинге 10.3 — `IEnumerable<string>`. Тип выходного элемента определяется последним выражением запроса — `select` или, в некоторых случаях, `group`. Однако не все выражения запроса имеют результатом `IEnumerable<T>`. Это зависит от того, каким провайдером LINQ вы пользуетесь. Лично я выбрал `IEnumerable<T>`, потому что использую LINQ to Objects.

Как C# узнал, что я хочу использовать LINQ to Objects? Это произошло из-за того, что в выражении `from` я использовал массив в качестве источника. В более общем смысле LINQ to Objects будет использоваться при указании в качестве источника любого `IEnumerable<T>`, если только не доступен более специализированный провайдер. Тем не менее это не объясняет, как C# изначально обнаруживает провайдеров и как он выбирает между ними. Чтобы понять это, вам необходимо узнать, что компилятор проделывает с выражением запроса.



При объявлении переменных, содержащих запросы LINQ, очень часто используется ключевое слово `var`. Это необходимо, если `select` выдает экземпляры анонимного типа, потому что нет способа записать имя типа результирующего запроса. Но даже без учета анонимных типов `var` по-прежнему широко используется, и для этого имеются две причины. Во-первых, это просто вопрос согласованности: некоторые люди считают, что, поскольку вы должны использовать `var` для некоторых запросов LINQ, вам следует использовать его для всех. Другим аргументом выступает то, что типы запросов LINQ часто имеют подробные и некрасивые имена, а `var` делает код менее загроможденным. Это может стать насущной проблемой, связанной с ограниченной шириной полосы книги, поэтому во многих примерах в этой главе я отошел от своего обычного предпочтения явных типов и использовал `var`, чтобы все умещалось.

Как разворачиваются выражения запроса

Компилятор преобразует все выражения запроса в один или несколько вызовов методов. После этого с помощью тех же самых механизмов, которые C# использует для любого другого вызова метода, выбирается провайдер LINQ. Компилятор не имеет никакой встроенной концепции, на которой основаны провайдеры LINQ. В этом он полагается на соглашение. В листинге 10.4 показано, что делает компилятор с выражением запроса из листинга 10.3.

Листинг 10.4. Суть выражения запроса

```
IEnumerable<string> commaCultures =  
    CultureInfo.GetCultures(CultureTypes.AllCultures)  
    .Where(culture => culture.NumberFormat.NumberDecimalSeparator == ",")  
    .Select(culture => culture.Name);
```

Методы `Where` и `Select` являются примерами операторов LINQ. Оператор LINQ — это не что иное, как метод, который соответствует одному из стандартных шаблонов. Я опишу эти шаблоны на с. 538, в разделе «Стандартные операторы LINQ».

Код в листинге 10.4 — это один оператор, так что я объединил вызовы методов. Язываю метод `Where` для возвращаемого значения `GetCultures` изываю метод `Select` для возвращаемого значения `Where`. Форматирование выглядит немного своеобразно, но оно слишком длинное для одной строки. Хотя это не очень элегантно, я предпочитаю ставить «`.`» в начале строки при разделении связанных вызовов на несколько строк, потому что

так намного легче увидеть, что каждая новая строка продолжается с того места, где закончилась предыдущая. Точка в конце предыдущей строки выглядит аккуратнее, но вместе с тем значительно упрощает неправильное прочтение кода.

Компилятор превратил выражения `where` и `select` в лямбды. Обратите внимание, что переменная диапазона становится параметром в каждой лямбде. Это одна из причин того, почему не следует воспринимать аналогию между выражениями запросов и циклами `foreach` слишком буквально. В отличие от итерационной переменной `foreach`, переменная диапазона не существует в виде одной обычной переменной. В запросе это просто идентификатор, представляющий элемент из источника, и при разворачивании запроса в вызовы методов C# может в конечном итоге создать несколько реальных переменных для одной переменной диапазона, как это происходит здесь с аргументами для двух отдельных лямбд.

Все выражения запросов сводятся к связанным вызовам методов с лямбдами. (Вот почему нам не обязательно нужен синтаксис выражения запроса — вы можете написать любой запрос, используя вызовы методов.) Некоторые из них сложнее, чем другие. Выражение в листинге 10.1 превращается в более простую структуру, несмотря на то что выглядит почти идентично листингу 10.3. Листинг 10.5 показывает, как оно разворачивается. Оказывается, что когда выражение `select` запроса напрямую передает переменную диапазона, компилятор интерпретирует это так, как будто мы хотим передать результаты предыдущего выражения напрямую без дальнейшей обработки, и не добавляет вызов `Select`. (Из этого есть одно исключение: если вы напишете выражение запроса, которое не содержит ничего, кроме выражений `from` и `select`, он генерирует вызов `Select`, даже если выражение `select` пустое.)

Листинг 10.5. Как разворачиваются пустые выражения `select`

```
IEnumerable<CultureInfo> commaCultures =
    CultureInfo.GetCultures(CultureTypes.AllCultures)
    .Where(culture => culture.NumberFormat.NumberDecimalSeparator == ",");
```

Компилятору придется потрудиться, если вы добавите в область действия запроса несколько переменных. Это можно сделать с помощью выражения `let`. Листинг 10.6 выполняет ту же работу, что и листинг 10.3, но в нем я добавил новую переменную `numFormat` для ссылки на числовой формат. Это делает мое выражение `where` более коротким и легким для чтения. В более сложном запросе с многократными ссылками на этот объект формата эта техника помогла бы устраниить хаос.

Листинг 10.6. Запрос с выражением let

```
IEnumerable<string> commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    let numFormat = culture.NumberFormat
    where numFormat.NumberDecimalSeparator == ","
    select culture.Name;
```

Когда вы пишете подобный запрос, который добавляет дополнительные переменные, компилятор автоматически генерирует скрытый класс с полем для каждой из переменных, чтобы сделать их доступными на каждом этапе. Чтобы получить тот же эффект с обычными вызовами методов, нам нужно сделать что-то подобное, и самый простой способ сделать это — добавить анонимный тип для их хранения, как показано в листинге 10.7.

Листинг 10.7. Как разворачиваются (примерно) выражения запроса с многими переменными

```
IEnumerable<string> commaCultures =
    CultureInfo.GetCultures(CultureTypes.AllCultures)
    .Select(culture => new { culture, numFormat = culture.NumberFormat })
    .Where(vars => vars.numFormat.NumberDecimalSeparator == ",")
    .Select(vars => vars.culture.Name);
```

Независимо от простоты или сложности выражения запросов это просто специализированный синтаксис для вызовов методов. А это значит, что можно написать собственный источник для выражения запроса.

Поддержка выражений запросов

Поскольку компилятор C# попросту преобразует различные выражения запроса в вызовы методов, мы можем написать тип, который участвует в этих выражениях. Для этого требуется определить подходящие методы. Чтобы показать, что компилятору C# действительно все равно, что делают эти методы, в листинге 10.8 показан класс, который не имеет никакого смысла, но полностью удовлетворяет C#, когда используется из выражения запроса. Компилятор механически преобразует выражение запроса в серию вызовов методов, поэтому, если есть подходящие методы, код успешно компилируется.

Я могу использовать экземпляр этого класса в качестве источника выражения запроса. Это безумие, потому что этот класс никоим образом не представляет коллекцию данных, однако компилятору все равно. Ему просто нужны определенные методы, поэтому, если я напишу код из лис-

тинга 10.9, компилятор будет счастлив, даже если этот код совершенно бессмысленный.

Листинг 10.8. Бессмысленные выражения `Where` и `Select`

```
public class SillyLinqProvider
{
    public SillyLinqProvider Where(Func<string, int> pred)
    {
        Console.WriteLine("Where invoked");
        return this;
    }

    public string Select<T>(Func<DateTime, T> map)
    {
        Console.WriteLine($"Select invoked, with type argument
{typeof(T)}");
        return "This operator makes no sense";
    }
}
```

Листинг 10.9. Бессмысленный запрос

```
var q = from x in new SillyLinqProvider()
        where int.Parse(x)
        select x.Hour;
```

Компилятор преобразует это в вызовы методов точно так же, как проделал это с более осмысленным запросом в листинге 10.1. Листинг 10.10 показывает результат. Если вы будете внимательны, то заметите, что моя переменная диапазона фактически меняет тип на полпути — моему методу `Where` требуется делегат, который принимает строку, поэтому в первой лямбде `x` имеет тип `string`. Но мой метод `Select` требует, чтобы его делегат принимал `DateTime`, так что это и есть тип `x` в этой лямбде. (И все это в конечном итоге не имеет значения, потому что мои методы `Where` и `Select` даже не используют эти лямбды.) Опять же, это чепуха, но она показывает, насколько механически компилятор C# преобразует запросы в вызовы методов.

Листинг 10.10. Как компилятор преобразует бессмысленный запрос

```
var q = new SillyLinqProvider().Where(x => int.Parse(x)).Select(x =>
x.Hour);
```

Очевидно, что писать код, который не имеет смысла, так себе идея. Я делаю все это только для того, чтобы показать, что синтаксис выражения запроса ничего не знает о семантике — у компилятора нет особых ожиданий относи-

тельно того, что будет делать каждый из вызываемых им методов. Все, что ему требуется, — это чтобы они принимали лямбды в качестве аргументов и возвращали что-то отличное от `void`.

Очевидно, что настоящая работа происходит где-то в другом месте. Ее, собственно, и выполняют провайдеры LINQ. Так что сейчас я опишу, что нам нужно написать, чтобы показанные в первых двух примерах запросы работали безо всякого LINQ to Objects.

Вы видели, как запросы LINQ преобразуются в код, подобный показанному в листинге 10.4, но это еще не все. Выражение `where` становится вызовом метода `Where`, но мы вызываем его для массива типа `CultureInfo[]`, который на самом деле не имеет метода `Where`. Это работает лишь потому, что LINQ to Objects определяет соответствующий метод расширения. Как я показал в главе 3, к существующим типам можно добавлять новые методы, что LINQ to Objects и делает в отношении `IEnumerable<T>`. (Так как большинство коллекций реализуют `IEnumerable<T>`, LINQ to Objects можно использовать практически для любого типа коллекций.) Чтобы использовать эти методы расширения, вам нужна директива `using` для пространства имен `System.Linq`. (Кстати, все методы расширения определяются статическим классом в этом пространстве имен с именем `Enumerable`.) Если попытаться использовать LINQ без этой директивы, компилятор выдаст следующую ошибку для выражения запроса в листинге 10.1 или 10.3:

```
error CS1935: Could not find an implementation of the query pattern
for source type 'System.Globalization.CultureInfo[]'. 'Where' not
found. Are you missing a reference to 'System.Core.dll' or a using
directive for 'System.Linq'?
```

В целом это сообщение об ошибке весьма полезно, но в нашем случае я хочу написать собственную реализацию LINQ¹. Листинг 10.11 содержит ее, и я показал весь исходный файл, потому что методы расширения чувствительны к использованию пространств имен и директив `using`. Содержимое метода `Main` должно показаться знакомым — это код из листинга 10.3, но на этот раз вместо использования провайдера LINQ to Objects он будет использовать методы расширения из моего класса `CustomLinqProvider`. (Обычно методы расширения делаются доступными посредством директивы `using`,

¹ Иногда полезно использовать направляющее указание. Рекомендация `System.Core.dll` верна только для .NET Framework, поскольку соответствующий код находится в `System.Linq.dll` в .NET Core, который в любом случае включен в ссылки, которые вы получаете по умолчанию в приложениях .NET Core.

но поскольку `CustomLinqProvider` находится в том же пространстве имен, что и класс `Program`, все его методы расширения автоматически доступны для `Main()`.



Хотя листинг 10.11 ведет себя как задумано, его нельзя воспринимать в качестве примера того, как провайдер LINQ обычно выполняет свои запросы. Он показывает место провайдеров LINQ в этой картине, но, как я покажу позже, с тем, как этот код обрабатывает запрос, есть кое-какие проблемы. Кроме того, пример довольно минималистичен — в LINQ содержится далеко не только `Where` и `Select`, а большинство настоящих провайдеров предлагают чуть больше этих двух операторов.

Листинг 10.11. Пользовательский провайдер LINQ для `CultureInfo[]`

```
using System;
using System.Globalization;

namespace CustomLinqExample
{
    public static class CustomLinqProvider
    {
        public static CultureInfo[] Where(this CultureInfo[] cultures,
                                         Predicate<CultureInfo> filter)
        {
            return Array.FindAll(cultures, filter);
        }

        public static T[] Select<T>(this CultureInfo[] cultures,
                                    Func<CultureInfo, T> map)
        {
            var result = new T[cultures.Length];
            for (int i = 0; i < cultures.Length; ++i)
            {
                result[i] = map(cultures[i]);
            }
            return result;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
```

```
var commaCultures =
    from culture in CultureInfo.GetCultures(
        CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture.Name;

    foreach (string cultureName in commaCultures)
    {
        Console.WriteLine(cultureName);
    }
}
}
```

Как вы теперь хорошо знаете, выражение запроса в `Main` будет сначала вызывать `Where` для источника, а затем `Select` для того, что возвращает `Where`. Как и прежде, источником выступает возвращаемое значение `GetCultures`, которое является массивом типа `CultureInfo[]`. Это тип, для которого `CustomLinqProvider` определяет методы расширения, так что код вызовет `CustomLinqProvider.Where`.

При этом для поиска всех элементов исходного массива, которые соответствуют предикату, используется метод `FindAll` класса `Array`. Метод `Where` передает свой собственный аргумент в качестве предиката напрямую в `FindAll`, и, как вы знаете, при вызове `Where` компилятор C# передает основанную на выражении лямбду в выражение `where` запроса LINQ. Этот предикат будет соответствовать культурам, которые используют в качестве десятичного разделителя запятую, поэтому выражение `Where` возвращает массив типа `CultureInfo[]`, который содержит только эти культуры.

Затем код, созданный компилятором из запроса, вызовет `Select` в массиве `CultureInfo[]`, возвращаемом `Where`. Массивы не имеют метода `Select`, поэтому будет использоваться метод расширения в `CustomLinqProvider`. Мой метод `Select` является обобщенным, поэтому компилятору нужно будет решить, каким должен быть аргумент типа, и он может понять это из выражения в выражении `select`.

Сначала компилятор преобразует его в лямбду: `culture => culture.Name`. Поскольку она станет вторым аргументом для `Select`, компилятор знает, что нам нужен `Func<CultureInfo, T>`, следовательно, параметр `culture` должен иметь тип `CultureInfo`. Это позволяет сделать вывод, что `T` должен быть типа `string`, потому что лямбда возвращает `culture.Name`, а тип этого

свойства `Name` — `string`. Таким образом, компилятор определяет, что он вызывает `CustomLinqProvider.Select<string>`. (Кстати, процесс решения, который я только что описал, не относится к имеющимся здесь выражениям запросов. Решение касательно типа происходит после того, как запрос был преобразован в вызовы метода. Компилятор прошел бы точно такой же путь, если бы мы начали с кода в листинге 10.4.)

Метод `Select` теперь создаст массив типа `string[]` (потому что здесь `T` — это `string`). Он заполняет этот массив путем перебора элементов входящего `CultureInfo[]`, передавая каждый `CultureInfo` в качестве аргумента лямбды, извлекающей свойство `Name`. Так мы получаем массив строк, содержащий название каждой культуры, которая использует запятую в качестве десятичного разделителя.

Это более реалистичный пример, нежели мой `SillyLinqProvider`, потому что теперь он обеспечивает ожидаемое поведение. Но, хотя запрос выдает те же строки, что и при использовании реального провайдера LINQ to Objects, механизм, с помощью которого он это делает, немного отличается. Мой `CustomLinqProvider` выполнял каждую операцию немедленно — оба метода, `Where` и `Select`, возвращали полностью заполненные массивы. LINQ to Objects делает все совершенно иначе. Кстати, как и большинство других провайдеров LINQ.

Отложенное вычисление

Если бы LINQ to Objects работал так же, как мой пользовательский провайдер в листинге 10.11, он бы не справился с листингом 10.12. В нем имеется метод Фибоначчи, который возвращает бесконечную последовательность, — он будет выдавать числа из ряда Фибоначчи, пока код продолжает их запрашивать. Я использовал `IEnumerable<BigInteger>`, возвращаемый этим методом, в качестве источника для выражения запроса. Поскольку в самом начале у нас имеется директива `using System.Linq`, я снова использую LINQ to Objects.

Здесь используется метод расширения `Where`, который LINQ to Objects предоставляет для `IEnumerable<T>`. Если бы это работало так же, как метод `Where` моего класса `CustomLinqExtension` для `CultureInfo[]`, программа никогда бы не смогла показать ни единого числа. Мой метод `Where` не завершался до тех пор, пока не отбирал весь свой ввод и не создавал полностью заполненный

массив в качестве вывода. Если бы метод LINQ to Objects Where попытался проделать это с моим бесконечным перечислителем Фибоначчи, он бы никогда не завершился.

Листинг 10.12. Запрос с бесконечной последовательностью

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics;

class Program
{
    static IEnumerable<BigInteger> Fibonacci()
    {
        BigInteger n1 = 1;
        BigInteger n2 = 1;
        yield return n1;
        while (true)
        {
            yield return n2;
            BigInteger t = n1 + n2;
            n1 = n2;
            n2 = t;
        }
    }

    static void Main(string[] args)
    {
        var evenFib = from n in Fibonacci()
                      where n % 2 == 0
                      select n;

        foreach (BigInteger n in evenFib)
        {
            Console.WriteLine(n);
        }
    }
}
```

На самом деле листинг 10.12 работает превосходно — он создает устойчивый поток вывода, состоящий из чисел Фибоначчи, которые делятся на 2. Это означает, что у него не получится выполнить всю выборку, когда вызывается `Where`. Вместо этого его метод `Where` возвращает `IEnumerable<T>`, который отбирает элементы по требованию. Он не будет пытаться извлечь что-либо

из входной последовательности, пока кто-то не запросит значение, после чего начнет извлекать из источника одно значение за другим, пока делегат фильтра не сообщит, что совпадение найдено. После этого он вернет найденный элемент и не будет пытаться извлечь что-либо еще из источника, пока не будет запрошен следующий элемент. В листинге 10.13 показано, как можно реализовать это поведение с помощью функции C# `yield return`.

Листинг 10.13. Пользовательский отложенный оператор `Where`

```
public static class CustomDeferredLinqProvider
{
    public static IEnumerable<T> Where<T>(this IEnumerable<T> src,
                                            Func<T, bool> filter)
    {
        foreach (T item in src)
        {
            if (filter(item))
            {
                yield return item;
            }
        }
    }
}
```

Настоящая реализация `Where` в LINQ to Objects несколько сложнее. Он определяет особые случаи, такие как массивы и списки, и обрабатывает их способом, который несколько более эффективен, чем реализация общего назначения, к которой он прибегает для других типов. Однако принцип одинаков для `Where` и всех остальных операторов: эти методы не выполняют указанную работу. Вместо этого они возвращают объекты, которые будут выполнять работу по требованию. Что-то происходит лишь тогда, когда вы пытаетесь получить результаты запроса. Это называется отложенным вычислением.

Преимущество отложенного вычисления заключается в том, что оно не выполняется до тех пор, пока оно вам не понадобится, что позволяет работать с бесконечными последовательностями. Однако недостатки у него также имеются. Вам следует быть осторожным, чтобы избежать повторных запросов на вычисление. Листинг 10.14 демонстрирует эту ошибку, что приводит к лишней работе. Листинг перебирает несколько разных чисел и записывает каждое из них, используя формат валюты каждой культуры, которая использует запятую в качестве десятичного разделителя.



Если вы запустите код в Windows, то обнаружите, что большинство строк, которые отображает этот код, содержат символы ?, указывающие на то, что консоль не способна отображать большинство значков валюты. На самом деле она это умеет, просто ей необходимо разрешение. По умолчанию, для обеспечения обратной совместимости, консоль Windows использует 8-битную кодовую страницу. Если вы выполните команду `chcp 65001` из командной строки, она переключит окно консоли на кодовую страницу UTF-8, которая позволяет отображать любые символы Юникод, поддерживаемые выбранным вами консольным шрифтом. Возможно, есть смысл настроить консоль на использование Consolas или Lucida Console, чтобы максимально использовать это преимущество.

Листинг 10.14. Случайное повторное вычисление отложенного запроса

```
var commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture;

object[] numbers = { 1, 100, 100.2, 10000.2 };

foreach (object number in numbers)
{
    foreach (CultureInfo culture in commaCultures)
    {
        Console.WriteLine(string.Format(culture, "{0}: {1:c}",
            culture.Name, number));
    }
}
```

Проблема этого кода в том, что хотя переменная `commaCultures` инициализируется вне числового цикла, мы выполняем ее итерацию для каждого числа. А поскольку LINQ to Objects использует отложенные вычисления, это означает, что фактическое выполнение запроса переделывается каждый проход внешнего цикла. Таким образом, вместо однократного вычисления выражения `where` для каждой культуры (841 раз для моей системы) он выполняется четыре раза для каждой культуры (всего 3364 раза), поскольку запрос целиком выполняется один раз для каждого из четырех элементов в массиве `numbers`. Это не катастрофа — код по-прежнему работает правильно. Но если вы допустите такое в программе на сильно загруженном сервере, это повредит его пропускной способности.

Если вы знаете, что вам придется многократно перебирать результаты запроса, подумайте об использовании методов расширения `ToList` или `ToArray`, предоставляемых LINQ to Objects. Они выполняют весь запрос сразу, создавая `IList<T>` или массив `T[]` соответственно (очевидно, вы не должны использовать эти методы для бесконечных последовательностей). Затем вы можете проходить по нему столько раз, сколько захотите, не неся при этом никаких дополнительных затрат (помимо минимальных на чтение элементов массива или списка). Но в тех случаях, когда вы перебираете запрос только один раз, лучше их не использовать, так как они потребляют больше памяти, чем необходимо.

LINQ, обобщения и `IQueryable<T>`

Большинство провайдеров LINQ используют обобщенные типы. Это нигде не предписывается, но используется повсеместно. LINQ to Objects использует `IEnumerable<T>`. Некоторые провайдеры баз данных используют тип под названием `IQueryable<T>`. В более широком смысле шаблон должен иметь некоторый общий тип `Source<T>`, где `Source` — это некоторый источник элементов, а `T` — тип отдельного элемента. Тип источника с поддержкой LINQ делает доступными методы оператора в `Source<T>` для любого `T`, и эти операторы также обычно возвращают `Source<TResult>`, где `TResult` может соответствовать, а может и не соответствовать типу `T`.

`IQueryable<T>` интересен тем, что он предназначен для использования несколькими провайдерами. Этот интерфейс, его базовый интерфейс `IQueryable` и связанный с ним `IQueryProvider` показаны в листинге 10.15.

Листинг 10.15. `IQueryable` и `IQueryable<T>`

```
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}

public interface IQueryable<out T> : IEnumerable<T>, IQueryable
{
}

public interface IQueryProvider
```

```
{  
    IQueryables CreateQuery(Expression expression);  
    IQueryables<TElement> CreateQuery<TElement>(Expression expression);  
    object Execute(Expression expression);  
    TResult Execute<TResult>(Expression expression);  
}
```

Наиболее очевидная особенность `IQueryables<T>` состоит в том, что он не добавляет членов к своим базовым интерфейсам. Он предназначен для использования исключительно через методы расширения. Пространство имен `System.Linq` определяет все стандартные операторы LINQ для `IQueryables<T>` как методы расширения, предоставляемые классом `Queryable`. Однако все они просто обращаются к свойству `Provider`, определенному базовым `IQueryables`. Итак, в отличие от LINQ to Objects, где методы расширения `IEnumerables<T>` определяют поведение, реализация `IQueryables<T>` может сама решать, как обрабатывать запросы, потому что она предоставляет `IQueryProvider`, который и выполняет реальную работу.

Тем не менее все провайдеры LINQ на основе `IQueryables<T>` имеют одну общую черту: они интерпретируют лямбды как объекты выражений, а не делегаты. В листинге 10.16 показано объявление методов расширения `Where`, определенных для `IEnumerables<T>` и `IQueryables<T>`. Сравните параметры `predicate`.

Листинг 10.16. `Enumerable` против `Queryable`

```
public static class Enumerable  
{  
    public static IEnumerables<TSource> Where<TSource>(  
        this IEnumerables<TSource> source,  
        Func<TSource, bool> predicate)  
    ...  
}  
  
public static class Queryable  
{  
    public static IQueryables<TSource> Where<TSource>(  
        this IQueryables<TSource> source,  
        Expression<Func<TSource, bool>> predicate)  
    ...  
}
```

Расширение `Where` для `IEnumerables<T>` (LINQ to Objects) принимает функцию `<TSource, bool>`, и, как вы видели в главе 9, это тип делегата. Но метод

расширения `Where` для `IQueryable<T>` (используется многочисленными провайдерами LINQ) принимает `<Func<TSource, bool>`, и, как вы, опять же, видели в главе 9, компилятор создает объектную модель выражения и передает ее в качестве аргумента.

Провайдер LINQ обычно использует `IQueryable<T>`, если ему требуются эти деревья выражений. Обычно это происходит потому, что далее он будет проверять ваш запрос и преобразовывать его во что-то еще, например в запрос SQL.

Есть несколько других общих обобщенных типов, которые появляются в LINQ. Некоторые функции LINQ гарантируют создание элементов в определенном порядке, а некоторые – нет. Несколько операторов создают элементы в порядке, который зависит от порядка их ввода. Это может быть отражено в типах, для которых определены операторы, и в типах, которые они возвращают. LINQ to Objects определяет `IOrderedEnumerable<T>` для упорядоченных данных, и есть соответствующий тип `IOrderedQueryable<T>` для провайдеров на основе `IQueryable<T>`. (Провайдеры, которые используют собственные типы, как правило, делают нечто подобное. Например, `Parallel LINQ` (см. главу 16) определяет запрос `OrderedParallel<T>`.) Эти интерфейсы являются производными от неупорядоченных аналогов, таких как `IEnumerable<T>` и `IQueryable<T>`. Таким образом, все обычные операторы доступны, но можно определять операторы или другие методы, которые учитывают существующий порядок их ввода. Например, в разделе «Упорядоченное расположение» на с. 551 я покажу оператор LINQ под названием `ThenBy`, который доступен только для уже упорядоченных источников.

При рассмотрении LINQ to Objects это разделение на упорядоченное/неупорядоченное может показаться лишним, потому что `IEnumerable<T>` всегда выдает элементы в каком-то порядке. Но некоторые провайдеры не обязательно соблюдают какой-то определенный порядок. Возможно, они распараллеливают выполнение запроса или заставляют выполнять запрос базу данных, а базы данных оставляют за собой право вмешиваться в порядок в случаях, когда это позволяет им работать более эффективно.

Стандартные операторы LINQ

В этом разделе я опишу стандартных операторов, которые могут содержать провайдеры LINQ. Там, где это возможно, я также опишу эквивалент выра-

жения запроса, хотя многие операторы не имеют соответствующей формы выражения запроса. Некоторые функции LINQ доступны только через явный вызов метода. Это верно даже для некоторых операторов, которые могут использоваться в выражениях запросов, поскольку большинство операторов перегружены, а выражения запросов не могут использовать некоторые из самых сложных перегрузок.



Операторы LINQ не являются операторами в обычном понимании C# – это не символы вроде + или &&. LINQ имеет собственную терминологию, и в рамках этой главы обозначим оператор как возможность запроса, предлагаемого провайдером LINQ. В C# это выглядит как метод.

У этих операторов есть что-то общее: все они спроектированы для поддержки композиции. Это означает, что вы можете комбинировать их практически любым удобным для вас способом, что позволяет создавать сложные запросы из простых элементов. Для этого операторы не только принимают некоторый тип, представляющий собой набор элементов (например, `IEnumerable<T>`), но большинство из них также и возвращают что-то, представляющее собой набор элементов. Как уже упоминалось, тип этих элементов не всегда один и тот же – оператор может взять `IEnumerable<T>` в качестве ввода и вернуть `IEnumerable<TResult>` в качестве вывода, где `TResult` не всегда совпадает с `T`. Тем не менее вы все равно можете все это увязать множеством разных способов. Частично это работает потому, что операторы LINQ подобны математическим функциям в том смысле, что они не изменяют свои входные данные; скорее они производят новый результат, который основан на их операндах. (Этим обычно характеризуются функциональные языки программирования.) Это означает, что вы можете не только объединять операторы в произвольные комбинации, не опасаясь побочных эффектов, но также использовать один и тот же источник в качестве входных данных для нескольких запросов, поскольку ни один из них не изменяет свой ввод. Каждый оператор возвращает новый запрос на основе своего ввода.

Этот функциональный стиль ничем не навязывается. Как вы видели в случае моего `SillyLinqProvider`, компилятору не важно, что делает метод, представляющий оператор LINQ. Тем не менее соглашение заключается в том, что операторы остаются функциональными, чтобы поддерживать возможность композиции. Все встроенные провайдеры LINQ работают таким образом.

Не все провайдеры предлагают полную поддержку всех операторов. Основные провайдеры от Microsoft, такие как LINQ to Objects или поддержка LINQ в Entity Framework и Rx, настолько исчерпывающие, насколько это возможно, но в ряде ситуаций некоторые операторы не имеют смысла.

Чтобы продемонстрировать операторы в действии, нужны исходные данные. Многие из дальнейших примеров будут использовать код из листинга 10.17.

Листинг 10.17. Пример входных данных для запросов LINQ

```
public class Course
{
    public string Title { get; set; }

    public string Category { get; set; }

    public int Number { get; set; }

    public DateTime PublicationDate { get; set; }

    public TimeSpan Duration { get; set; }

    public static readonly Course[] Catalog =
    {
        new Course
        {
            Title = "Elements of Geometry",
            Category = "MAT", Number = 101, Duration =
            TimeSpan.FromHours(3),
            PublicationDate = new DateTime(2009, 5, 20)
        },
        new Course
        {
            Title = "Squaring the Circle",
            Category = "MAT", Number = 102, Duration =
            TimeSpan.FromHours(7),
            PublicationDate = new DateTime(2009, 4, 1)
        },
        new Course
        {
            Title = "Recreational Organ Transplantation",
            Category = "BIO", Number = 305, Duration =
            TimeSpan.FromHours(4),
            PublicationDate = new DateTime(2002, 7, 19)
        },
    }
```

```
new Course
{
    Title = "Hyperbolic Geometry",
    Category = "MAT", Number = 207, Duration =
    TimeSpan.FromHours(5),
    PublicationDate = new DateTime(2007, 10, 5)
},
new Course
{
    Title = "Oversimplified Data Structures for Demos",
    Category = "CSE", Number = 104, Duration =
    TimeSpan.FromHours(2),
    PublicationDate = new DateTime(2019, 9, 21)
},
new Course
{
    Title = "Introduction to Human Anatomy and Physiology",
    Category = "BIO", Number = 201, Duration =
    TimeSpan.FromHours(12),
    PublicationDate = new DateTime(2001, 4, 11)
},
);
}
```

Фильтрация

Один из самых простых операторов — это `Where`, который фильтрует входные данные. Вы предоставляете предикат, который является функцией, принимающей отдельный элемент и возвращающей `bool`. `Where` возвращает объект, содержащий элементы из входных данных, для которых предикат имеет значение `true`. (Концептуально это очень похоже на метод `FindAll`, доступный в `List<T>` и типах массивов, но с использованием отложенного выполнения.)

Как вы уже видели, в запросах этот оператор представлен с помощью выражения `where`. Но у оператора `Where` есть перегрузка, которая предоставляет дополнительную функцию, недоступную из выражения запроса. Вы можете написать лямбда-фильтр, принимающий два аргумента: элемент из входных данных и индекс, указывающий позицию этого элемента в источнике. В листинге 10.18 эта форма используется для удаления каждого второго числа из входных данных, а также для удаления курсов продолжительностью менее трех часов.

Листинг 10.18. Оператор `Where` с индексом

```
IEnumerable <Course> q = Course.Catalog.Where (
    (course, index) => (index % 2 == 0) &&
    course.Duration.TotalHours >= 3);
```

Индексированная фильтрация имеет смысл только для упорядоченных данных. С `LINQ to Objects` она работает всегда, потому что там используется `IEnumerable<T>`, который выдает элементы один за другим. Но не все провайдеры LINQ обрабатывают элементы последовательно. Например, в `Entity Framework` запросы LINQ, которые вы пишете на C#, будут обрабатываться в базе данных. Если запрос явно не запрашивает какой-то конкретный порядок, база данных может обрабатывать элементы в любом порядке по своему усмотрению, иногда параллельно. В некоторых случаях база данных может использовать стратегии оптимизации, которые позволяют получать требуемые результаты, используя процесс, который мало похож на исходный запрос. Так что может оказаться бессмысленным говорить, скажем, о 14-м элементе, который обрабатывается выражением `WHERE`. Следовательно, если бы вы написали запрос, аналогичный листингу 10.18, но с использованием `Entity Framework`, то выполнение запроса вызвало бы исключение с сообщением о том, что индексированный оператор `Where` неприменим. Если вам интересно, почему перегрузка присутствует, даже если провайдер ее не поддерживает, то ответ в том, что `Entity Framework` использует `IQueryable<T>`, для которого во время компиляции доступны все операторы. Провайдеры, которые используют `IQueryable<T>`, могут лишь во время выполнения сообщить о недоступности операторов.

Вы, возможно, ожидали, что исключение возникнет при вызове `Where`, а не при попытке выполнить запрос (т. е. при первой попытке получить один или несколько элементов). Однако провайдеры, которые преобразуют запросы LINQ в какую-либо другую форму, такую как запрос SQL, обычно откладывают проверку до выполнения запроса. Это связано с тем, что некоторые операторы могут быть действительными только в определенных сценариях, а это означает, что провайдер до построения всего запроса не может знать, будет ли работать какой-либо конкретный оператор. Поведение, при котором ошибки, вызванные нежизнеспособными запросами, иногда возникали бы при построении запроса, а иногда — при его выполнении, было бы непоследовательным. Поэтому даже в тех случаях, когда провайдер способен заранее определить неудачный запрос, он обычно откладывает сообщение об этом до момента его вызова.



Провайдеры LINQ, которые реализуют часть или всю логику запросов на стороне сервера, обычно накладывают ограничения на то, что вы можете делать в составляющих запрос лямбдах. И наоборот, LINQ to Objects выполняет запросы в процессе, поэтому позволяет вам вызывать любой метод из лямбда-фильтра. Если вы хотите вызвать `Console.WriteLine` или прочитать в предикате данные из файла, LINQ to Objects не сможет вас остановить. Но в провайдерах для баз данных доступен очень ограниченный выбор методов. Эти провайдеры должны иметь возможность переводить ваши лямбды в то, что сервер способен обработать, и они будут отклонять выражения, которые пытаются вызвать методы, не имеющие эквивалента на стороне сервера.

Лямбда-фильтр оператора `Where` должен принимать аргумент типа элемента (например, `T` в `IEnumerable<T>`), а возвращать `bool`. Из главы 9 вы можете помнить, что библиотека классов определяет подходящий тип делегата под названием `Predicate<T>`, но я также упоминал, что LINQ старается его избегать, и теперь можно понять почему. Индексированная версия оператора `Where` не может использовать `Predicate<T>` из-за дополнительного аргумента, так что перегрузка использует `Func<T, int, bool>`. Ничто не мешает неиндексированной форме `Where` использовать `Predicate<T>`, но провайдеры LINQ, как правило, всегда используют `Func`, чтобы гарантировать, что операторы с одинаковым смыслом имеют похожие сигнатуры. Поэтому в целях согласованности большинство провайдеров используют `Func<T, bool>`. (C# не волнует, что именно вы используете, — выражения запросов по-прежнему работают, если провайдер использует `Predicate<T>`. Это иллюстрация моего пользовательского оператора `Where` в листинге 10.11, но ни один из провайдеров Microsoft так не работает.)

В LINQ имеется еще один оператор фильтрации: `OfType<T>`. Он нужен, когда ваш источник содержит смесь различных типов элементов — возможно, источником является `IEnumerable<object>` и вы хотели бы отобрать из него только элементы типа `string`. Листинг 10.19 показывает, как это можно сделать с использованием оператора `OfType<T>`.

Листинг 10.19. Оператор `OfType<T>`

```
static void ShowAllStrings(IEnumerable<object> src)
{
    foreach (string s in src.OfType<string>())
    {
        Console.WriteLine(s);
    }
}
```

Как `Where`, так и `OfType<T>` будет выдавать пустые последовательности, если ни один из объектов в источнике не соответствует требованиям. Это не считается ошибкой — пустые последовательности вполне допустимы в LINQ. Многие операторы могут выдавать их в качестве вывода, и большинство операторов могут принимать их в качестве ввода.

Select

При написании запроса нам может потребоваться извлечь из исходных элементов только определенные фрагменты данных. Выражение `select` в конце большинства запросов позволяет нам указать лямбду, которая будет использоваться для создания конечных выходных элементов. Есть несколько причин, по которым может потребоваться, чтобы наше выражение `select` делало больше, чем просто напрямую передавало каждый элемент. Нам может понадобиться лишь определенный фрагмент информации от каждого элемента, или мы хотим полностью преобразовать его, превратив во что-то другое.

Вы уже видели несколько выражений `select`, и в листинге 10.3 я показал, что компилятор превращает их в вызов `Select`. Однако, как и во многих операторах LINQ, версия, доступная через выражение запроса, — не единственный вариант. Имеется еще одна перегрузка, которая предоставляет не только элемент ввода, из которого создается элемент вывода, но также и индекс этого элемента. Листинг 10.20 использует ее для создания нумерованного списка названий курсов.

Листинг 10.20. Оператор Select с индексом

```
IEnumerable<string> nonIntro = Course.Catalog.Select((course, index) =>
    $"Course {index}: {course.Title}");
```

Имейте в виду, что отсчитываемый от нуля индекс, передаваемый в лямбду, будет основан на том, что попадает в оператор `Select`, и не обязательно будет представлять исходную позицию элемента в лежащем ниже источнике данных. В коде, похожем на листинг 10.21, результаты могут отличаться от ожидаемых.

Листинг 10.21. Индексированный оператор Select далее в операторе Where

```
IEnumerable<string> nonIntro = Course.Catalog
    .Where(c => c.Number >= 200)
    .Select((course, index) => $"Course {index}: {course.Title}");
```

Этот код отберет в массиве `Course.Catalog` курсы по индексам 2, 3 и 5 соответственно, потому что это те курсы, свойство `Number` которых удовлетворяет выражению `Where`. Однако запрос пронумерует эти три курса как 0, 1 и 2, потому что оператор `Select` видит только те элементы, которые были прошлены выражением `Where`. Выражение `Select` никогда не имело доступа к исходному источнику, так что в его понимании есть только три элемента. Если вы хотите, чтобы индексы относились к исходной коллекции, нужно было бы извлечь их перед `Where`, как показано в листинге 10.22.

Листинг 10.22. Индексированное выражение `Select` выше оператора `Where`

```
IEnumerable<string> nonIntro = Course.Catalog
    .Select((course, index) => new { course, index })
    .Where(vars => vars.course.Number >= 200)
    .Select(vars => $"Course {vars.index}: {vars.course.Title}");
```

Вас может удивить, что я использовал здесь анонимный тип вместо кортежа. Я мог бы заменить `new {course, index}` на просто `(course, index)`, и код работал бы так же хорошо. Однако в целом кортежи не всегда работают в LINQ. Облегченный синтаксис кортежей был введен в C# 7.0, поэтому их не было, когда деревья выражений добавлялись в C# 3.0. Объектная модель выражения не обновлялась для поддержки этой языковой функции, поэтому, если вы попытаетесь использовать кортеж с провайдером LINQ на основе `IQueryable<T>`, вы получите ошибку компилятора CS8143, сообщающую, что «An expression tree may not contain a tuple literal» («Дерево выражений может не содержать литерал кортежа»). Поэтому в этой главе я стараюсь использовать анонимные типы, потому что они хорошо работают с провайдерами на основе запросов. Но если вы используете исключительно локальный провайдер LINQ (например, Rx или LINQ to Objects), смело действуйте кортежи.

Индексированный оператор `Select` аналогичен индексируемому оператору `Where`. Так что, как и следовало ожидать, не все провайдеры LINQ поддерживают его во всех сценариях.

Формирование данных и анонимные типы

Если вы используете провайдера LINQ для доступа к базе данных, оператор `Select` может уменьшить количество извлекаемых данных, что способно снизить нагрузку на сервер. Когда для выполнения запроса, который возвращает набор объектов, представляющих постоянные сущности, вы используете технологию доступа к данным, такую как `Entity Framework`, это всегда компромисс между выполнением слишком большого количества

предварительных задач и необходимостью выполнять много дополнительной отложенной работы. Должны ли эти структуры полностью заполнять все свойства объекта, которые соответствуют столбцам в различных таблицах базы данных? Должны ли они также загружать связанные объекты? В целом, более эффективно не извлекать данные, которые вы не собираетесь использовать, и данные, которые не извлекаются заранее, всегда могут быть загружены по требованию позже. Но, если вы проявляете излишнюю скромность в первоначальном запросе, вы можете в конечном итоге оказаться вынужденными сделать много дополнительных запросов для того, чтобы заполнить пробелы, а это может перевесить любую выгоду от попыток избежать ненужной работы.

Когда речь идет о связанных объектах, Entity Framework позволяет вам настроить, какие связанные объекты должны быть выбраны предварительно, а какие загружены по требованию, но для любого конкретного получаемого объекта все свойства, относящиеся к столбцам, обычно заполняются полностью. Это означает, что запросы, требующие целых объектов, в конечном итоге получают все столбцы любой строки, которой они касаются.

Если вам нужны только один или два столбца, это будет относительно дорого.

Листинг 10.23 использует этот несколько неэффективный подход. Он демонстрирует довольно типичный запрос Entity Framework.

Этот провайдер LINQ переводит выражение `where` в эффективный эквивалент SQL. Однако выражение SQL `SELECT` извлекает из таблицы все столбцы. Сравните это с листингом 10.24. В нем изменена только одна часть запроса: выражение `select` теперь возвращает экземпляр анонимного типа, который содержит только те свойства, которые нам необходимы. (Цикл, следующий за запросом, можно оставить нетронутым. Он использует `var` для своей итерационной переменной, которая будет отлично работать с анонимным типом, предоставляющим три свойства, требующиеся циклу.)

Листинг 10.23. Получение большего количества данных, чем необходимо

```
var pq = from product in dbCtx.Product
        where product.ListPrice > 3000
        select product;
foreach (var prod in pq)
{
    Console.WriteLine($"{prod.Name} ({prod.Size}): {prod.ListPrice}");
}
```

Листинг 10.24. Выражение select с анонимным типом

```
var pq = from product in dbCtx.Product
         where (product.ListPrice > 3000)
         select new { product.Name, product.ListPrice, product.Size };
```

Код дает точно такие же результаты, но генерирует гораздо более компактный SQL-запрос, который запрашивает только столбцы `Name`, `ListPrice` и `Size`. Если вы используете таблицу с множеством столбцов, это значительно сократит отклик, поскольку на него больше не влияют данные, которые нам не нужны. Это снижает нагрузку на сетевое подключение к серверу базы данных, а также приводит к более быстрой обработке, поскольку получение данных займет меньше времени. Эта техника называется формированием данных.

Этот подход не всегда является улучшением. С одной стороны, это означает, что вы работаете непосредственно с данными в базе данных, вместо того чтобы использовать объекты сущностей. Это может означать работу на более низком уровне абстракции, чем это было бы возможно при использовании типов сущностей, а это может увеличить затраты на разработку. Кроме того, в некоторых средах администраторы баз данных не разрешают специальные запросы, заставляя вас использовать хранимые процедуры, и в этом случае вы не сможете использовать эту технику.

Кстати, проекция результатов запроса в анонимный тип не ограничивается запросами к базе данных. Вы можете делать это с любым провайдером LINQ, например LINQ to Objects. Иногда это может оказаться полезным способом получения структурированной информации из запроса без необходимости специально определять класс. (Как я уже упоминал в главе 3, анонимные типы могут использоваться за пределами LINQ, но это один из основных сценариев, для которых они были разработаны. Другой — это группировка по составным ключам, которую я опишу в разделе «Группировка» на с. 570.)

Проекция и преобразование данных

Оператор `Select` иногда называют проекцией, и это та же самая операция, которую многие языки называют преобразованием данных. Это позволяет немного по-другому взглянуть на оператор `Select`. До сих пор я представлял `Select` как способ отбора того, что получится в результате запроса, но его можно рассматривать и как способ применить преобразование к каждому элементу в источнике. Для создания модифицированных версий списка чисел в листинге 10.25 используется `Select`. Он удваивает числа, возводит их в квадрат и превращает в строки.

Листинг 10.25. Использование Select для преобразования чисел

```
int[] numbers = { 0, 1, 2, 3, 4, 5 };

IEnumerable<int> doubled = numbers.Select(x => 2 * x);
IEnumerable<int> squared = numbers.Select(x => x * x);
IEnumerable<string> numberText = numbers.Select(x => x.ToString());
```

SelectMany

Оператор LINQ `SelectMany` используется в выражениях запросов, которые содержат несколько выражений `from`. Он называется `SelectMany`, потому что вместо выбора одного элемента вывода для каждого элемента ввода вы предоставляете ему лямбду, которая выдает для каждого элемента ввода целую коллекцию. Результирующий запрос выдает все объекты из всех этих коллекций, как если бы все коллекции, возвращаемые вашей лямбдой, были объединены в одну. (Эта операция не удалит дубликаты. В LINQ последовательности могут содержать дубликаты. Вы можете удалить их с помощью оператора `Distinct`, описанного в «Операции над множествами» на с. 567.) Этот оператор можно рассматривать с нескольких точек зрения. Одна из них заключается в том, что он дает способ объединения двух уровней иерархии — коллекции коллекций — в один. Другой способ — рассматривать его как на декартово произведение, т. е. способ получить любую возможную комбинацию из некоторых входных множеств.

В листинге 10.26 показано, как следует использовать этот оператор в выражении запроса. Этот код делает акцент на поведении, похожем на декартово произведение. Он показывает каждую комбинацию букв A, B и C и одной цифры от 1 до 5, т. е. A1, B1, C1, A2, B2, C2 и т. д. (Если вас волнует очевидная несовместимость двух входных последовательностей, выражение `select` этого запроса опирается на тот факт, что, если вы используете оператор `+` для сложения строки и какого-то другого типа, C# генерирует код, который за вас вызывает `ToString` нестрокового типа.)

Листинг 10.26. Использование `SelectMany` из выражения запроса

```
int[] numbers = { 1, 2, 3, 4, 5 };
string[] letters = { "A", "B", "C" };

IEnumerable<string> combined = from number in numbers
                                from letter in letters
                                select letter + number;

foreach (string s in combined)
```

```
{  
    Console.WriteLine(s);  
}
```

Листинг 10.27 показывает, как вызвать оператор напрямую, что эквивалентно выражению запроса в листинге 10.26.

Листинг 10.27. Оператор SelectMany

```
IEnumerable<string> combined =  
    numbers.SelectMany( number => letters,  
        (number, letter) => letter + number);
```

В листинге 10.26 используются две фиксированные коллекции: второе выражение `from` каждый раз возвращает одну и ту же коллекцию `letters`. Однако вы можете заставить выражение во втором выражении `from` возвращать значение, основанное на текущем элементе из первого выражения `from`. В листинге 10.27 вы можете видеть, что первая лямбда, переданная в `SelectMany` (которая фактически соответствует второму `from` последнего выражения), получает текущий элемент из первой коллекции через аргумент `number`, так что вы можете использовать его для выбора новой коллекции для каждого элемента из первой коллекции. Таким образом можно использовать сглаживающий (flattening) эффект, который дает `SelectMany`.

Я скопировал ступенчатый массив из листинга 5.16 в главе 5 в листинг 10.28, который обрабатывает его с помощью запроса, содержащего два выражения `from`. Обратите внимание, что выражение во втором выражении `from` теперь `row`, переменная диапазона первого выражения `from`.

Листинг 10.28. Сглаживание ступенчатого массива

```
int[][] arrays =  
{  
    new[] { 1, 2 },  
    new[] { 1, 2, 3, 4, 5, 6 },  
    new[] { 1, 2, 4 },  
    new[] { 1 },  
    new[] { 1, 2, 3, 4, 5 }  
};  
  
IEnumerable<int> flattened = from row in arrays  
                                from number in row  
                                select number;
```

Первое выражение `from` просит выполнить итерацию по каждому элементу в массиве верхнего уровня. Каждый из этих элементов также является массивом, и второе выражение `from` просит выполнить итерацию для каждого из этих вложенных массивов. Тип этого вложенного массива — `int []`, поэтому переменная диапазона второго выражения `from`, т. е. `number`, представляет собой `int` из этого вложенного массива. Выражение `select` просто возвращает каждое из этих значений `int`.

Результирующая последовательность поочередно содержит каждое число в массивах. Это сгладило ступенчатый массив в простую линейную последовательность чисел. Концептуально такое поведение похоже на написание вложенной пары циклов, один из которых проходит по внешнему массиву `int[][]`, а другой — по содержимому каждого отдельного массива `int[]`.

Для листинга 10.28 компилятор использует ту же перегрузку `SelectMany`, что и для листинга 10.27, но в данном случае есть и альтернатива. Заключительное выражение `select` в листинге 10.28 проще — оно без изменений передает элементы из второй коллекции, что означает, что более простая перегрузка, показанная в листинге 10.29, выполнит эту работу так же хорошо. Используя ее, мы просто предоставляем одну лямбду, которая выбирает коллекцию, которую `SelectMany` развернет для каждого из элементов входной коллекции.

Листинг 10.29. SelectMany без проекции элемента

```
var flattened = arrays.SelectMany(row => row);
```

Это довольно лаконичный код, поэтому в случае, если не совсем понятно, как это может привести к сглаживанию массива, в листинге 10.30 показано, как можно реализовать `SelectMany` для `IEnumerable<T>` самостоятельно.

Листинг 10.30. Одна из реализаций SelectMany

```
static IEnumerable<T2> MySelectMany<T, T2>(
    this IEnumerable<T> src, Func<T, IEnumerable<T2>> getInner)
{
    foreach (T itemFromOuterCollection in src)
    {
        IEnumerable<T2> innerCollection = getInner(
            itemFromOuterCollection);
        foreach (T2 itemFromInnerCollection in innerCollection)
        {
            yield return itemFromInnerCollection;
        }
    }
}
```

Почему компилятор не использует более простой вариант, показанный в листинге 10.29? Спецификация языка C# определяет, как выражения запроса переводятся в вызовы методов, и упоминает только перегрузку, показанную в листинге 10.26. Возможно, причина, по которой в спецификации не упоминается более простая перегрузка, заключается в снижении требований, которые C# предъявляет к типам, поддерживающим эту форму запроса с двойным `from`. Для того чтобы включить этот синтаксис для ваших собственных типов, вам нужно написать только один метод. Однако различные провайдеры LINQ .NET оказываются более щедрыми и предоставляют эту более простую перегрузку разработчикам, которые предпочитают использовать операторы напрямую. Фактически некоторые провайдеры определяют еще две перегрузки: существуют версии обеих встреченных нами до сих пор форм `SelectMany`, которые также передают первой лямбде индекс элемента. (Конечно, следует иметь в виду обычные предостережения относительно индексированных операторов.)

Хотя листинг 10.30 и дает разумное представление о том, что делает `LINQ to Objects` в `SelectMany`, это не точная реализация. Для особых случаев там предусмотрена оптимизация. Более того, другие провайдеры могут использовать совершенно другие стратегии. Базы данных часто имеютстроенную поддержку декартовых произведений, поэтому некоторые провайдеры могут реализовать `SelectMany` в такой форме.

Упорядоченное расположение

Как правило, запросы LINQ не гарантируют выдачу элементов в каком-либо конкретном порядке, если вы его явно не определяете. Это можно сделать в выражении запроса с выражением `orderby`. Как показано в листинге 10.31, для этого вы указываете выражение, согласно которому элементы должны быть упорядочены, и направление. В результате будет получена коллекция курсов, упорядоченных по возрастанию даты публикации. Так повелось, что по умолчанию используется `ascending`, так что вы можете опустить этот классификатор. Как вы уже, наверное, догадались, для изменения порядка вы можете указать `descending`.

Листинг 10.31. Выражение запроса с выражением `orderby`

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending
        select course;
```

Компилятор преобразует выражение `orderby` в листинге 10.31 в вызов метода `OrderBy` и будет использовать `OrderByDescending`, если вы указали исходящий порядок сортировки. В случае типов источников, которые различают упорядоченные и неупорядоченные коллекции, эти операторы возвращают упорядоченный тип (например, `IOrderedEnumerable<T>` для LINQ to Objects и `IOrderedQueryable<T>` для провайдеров, основанных на `IQueryable<T>`).



В случае LINQ to Objects, прежде чем выдать какие-либо выходные элементы, эти операторы должны извлечь каждый элемент своего ввода. Восходящий `OrderBy` сможет определить, какой элемент вернуть первым, только когда найдет самый младший элемент, и он не будет знать наверняка, какой из них самый-самый, пока не просмотрит их все. Он по-прежнему использует отложенное вычисление — он ничего не будет делать, пока вы не запросите первый элемент. Но как только вы что-то запросите, он должен будет выполнить всю работу сразу. Некоторые провайдеры могут обладать дополнительными сведениями о данных, которые могут обеспечить более эффективные стратегии. (Например, база данных может использовать индекс для возврата значений в требуемом порядке.)

У операторов `OrderBy` и `OrderByDescending` из LINQ to Objects есть две перегрузки, только одна из которых доступна из выражения запроса. Если вы вызываете методы напрямую, вы можете указать дополнительный параметр типа `IComparer< TKey >`, где `TKey` — тип выражения, по которому сортируются элементы. Это может оказаться важно, если вы сортируете по строковому свойству, потому что есть несколько разных способов упорядочить текст. Может потребоваться выбрать один из них в зависимости от локали вашего приложения, или же вы можете указать способ, который не зависит от культуры, чтобы обеспечить согласованность во всех средах.

Выражение, которое определяет порядок в листинге 10.31, очень простое — оно лишь извлекает свойство `PublicationDate` из исходного элемента. Если хотите, вы можете написать более сложные выражения. Если вы используете провайдер, который переводит запрос LINQ во что-то другое, вы можете столкнуться с ограничениями. Если запрос выполняется в базе данных, вы можете ссылаться на другие таблицы — провайдер способен преобразовать выражение, например `product.ProductCategory.Name`, в подходящее объединение. Однако вы не сможете запустить в этом выражении какой-либо старый код, потому что это должно быть что-то, что база данных способна выпол-

нить. Но LINQ to Objects просто по одному разу вызывает выражение для каждого объекта, так что вы действительно можете вставить туда любой код.

Вы можете осуществлять сортировку по нескольким критериям. Этого не следует делать с помощью нескольких выражений `orderby`. Листинг 10.32 допускает эту ошибку.

Листинг 10.32. Как нельзя применять несколько критериев сортировки

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending
        orderby course.Duration descending // ПЛОХО! Может нарушить
                                            // предыдущий порядок
        select course;
```

Этот код упорядочивает элементы по дате публикации, а затем по продолжительности, но делает это как две отдельные и не связанные операции. Второе выражение `orderby` гарантирует только то, что результаты будут в порядке, указанном в этом выражении, и не гарантирует сохранения исходного порядка. Если вам на самом деле нужно, чтобы элементы были упорядочены по дате публикации, а любые элементы с одинаковой датой публикации упорядочены по убыванию, вам понадобится запрос из листинга 10.33.

Листинг 10.33. Несколько критериев сортировки в выражении запроса

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending,
                course.Duration descending
        select course;
```

LINQ определяет для этого отдельные операторы: `ThenBy` и `ThenByDes`. В листинге 10.34 показано, как добиться того же эффекта, что и в выражении запроса в листинге 10.33, но путем непосредственного вызова операторов LINQ. Для провайдеров LINQ, чьи типы различают упорядоченные и неупорядоченные коллекции, например `IOrderedQueryable<T>` или `IOrderedEnumerable<T>`, эти два оператора будут доступны только в упорядоченной форме. Если вы попытаетесь непосредственно вызвать `ThenBy` для `Course.Catalog`, компилятор сообщит об ошибке.

Листинг 10.34. Несколько критериев сортировки с помощью операторов LINQ

```
var q = Course.Catalog
        .OrderBy(course => course.PublicationDate)
        .ThenByDescending(course => course.Duration);
```

Можно заметить, что некоторые операторы LINQ сохраняют некоторую упорядоченность, даже если вы их об этом не просите. Например, LINQ to Objects обычно выдает элементы в том же порядке, в котором они появились во входных данных, если только вы не напишете запрос, который заставит его изменить порядок. Но это просто продукт разработки LINQ to Objects, и в целом вам не следует на него полагаться. Даже когда вы используете этот конкретный провайдер LINQ, нужно свериться с документацией и убедиться, что порядок гарантирован, а не является просто случайной реализацией. В большинстве случаев, если вам нужен порядок, следует написать запрос, который говорит об этом явно.

Проверка принадлежности

LINQ определяет различные стандартные операторы получения данных о том, что содержит коллекция. Некоторые провайдеры могут реализовать эти операторы без необходимости проверки каждого элемента. (Например, провайдер на основе базы данных может использовать выражение `WHERE`, а база данных — индекс, не обращаясь к каждому элементу.) Вы можете без ограничений использовать эти операторы, а дело провайдера — выяснить, есть ли на самом деле короткий путь.



В отличие от большинства операторов LINQ, в большинстве провайдеров они не возвращают ни коллекций, ни элементов из своих входных данных. Как правило, они возвращают `true` или `false`, а в некоторых случаях — количество. Rx является заметным исключением: его реализации этих операторов заключают `bool` или `int` в одноэлементную `IEnumerable<T>`, которая и является результатом. Это сделано, чтобы сохранить реактивный характер обработки в Rx.

Самый простой оператор — `Contains`. Вы передаете элемент, и некоторые провайдеры (включая LINQ to Objects) предоставляют перегрузку, которая принимает так же и `IEqualityComparer<T>`, что позволяет настроить способ определения оператором тождественности элемента в источнике и указанного элемента. `Contains` возвращает `true`, если источник содержит указанный элемент, и `false`, если это не так. (Если вы используете версию с одним аргументом с коллекцией, которая реализует `ICollection<T>` (что включает в себя все реализации `IList<T>`), LINQ to Objects обнаруживает это и его реализация `Contains` просто полагается на эту коллекцию. Если вы

используете коллекции, не относящиеся к `ICollection<T>`, или предоставляемые пользовательский блок сравнения, он вынужден проверять каждый элемент в коллекции.)

Если вместо поиска конкретного значения нужно узнать, содержит ли коллекция значения, удовлетворяющие конкретным критериям, вы можете использовать оператор `Any`. Он принимает предикат и возвращает `true`, если предикат истинен хотя бы для одного элемента в источнике. Если вам требуется узнать, сколько элементов соответствует определенным критериям, вы можете использовать оператор `Count`. Он также принимает предикат, но вместо `bool` возвращает `int`. Если вы работаете с очень большими коллекциями, диапазон `int` может оказаться недостаточным, и в таком случае можно использовать оператор `LongCount`, который возвращает 64-битное число. (Это может показаться излишним для большинства применений LINQ to Objects, но может иметь значение, когда коллекция находится в базе данных.)



Остерегайтесь кода вроде `if (q.Count ()> 0)`. Расчет точного количества может потребовать выполнения всего исходного запроса (в данном случае `q`), и в любом случае для этого потребуется больше работы, чем для запроса типа есть ли здесь что-нибудь? Если `q` ссылается на запрос LINQ, то запись `if (q.Any())`, вероятно, окажется более эффективной. (Тем не менее вне LINQ это не относится к коллекциям, подобным списку, где получение количества элементов — достаточно дешевая операция, которая может оказаться более эффективной, чем оператор `Any`.)

Операторы `Any`, `Count` и `LongCount` имеют перегрузки, которые не принимают никаких аргументов. Для `Any` такая перегрузка возвращает `true`, если источник содержит хотя бы один элемент, а для `Count` и `LongCount` — сообщает, сколько именно элементов содержит источник.

Родственным оператору `Any` является оператор `All`. Он не имеет перегрузок и принимает предикат, возвращая `true` тогда и только тогда, когда источник не содержит элементов, которые не соответствуют предикату. Я использовал неуклюжее двойное отрицание не просто так: `All` возвращает `true` при применении к пустой последовательности, потому что пустая последовательность, безусловно, не содержит элементов, которые не соответствуют предикату по той простой причине, что она вообще не содержит никаких элементов.

АСИНХРОННОЕ НЕМЕДЛЕННОЕ ВЫЧИСЛЕНИЕ

Хотя, как вы уже видели, большинство операторов LINQ откладывают выполнение, есть и некоторые исключения. Для большинства провайдеров LINQ операторы `Contains`, `Any` и `All` не создают оболочки для выдаваемого результата. (Например, в `LINQ to Objects` они возвращают `bool`, а не `IEnumerable<bool>`.) Иногда это означает, что указанные операторы вынуждены выполнять работу медленно. Например, провайдеру `LINQ Entity Framework`, прежде чем он сможет вернуть результат `bool`, необходимо отправить запрос в базу данных и дождаться ответа. То же самое касается `ToArrayList` и `ToDictionary`, которые выдают полностью заполненные коллекции вместо `IEnumerable<T>` или `IQueryable<T>`, которые способны выдавать результаты в будущем.

Как описано в главе 16, для медленных операций, подобных этим, характерна реализация асинхронного шаблона на основе задач (ТАР), что позволяет нам использовать ключевое слово `await`, описанное в главе 17. Поэтому некоторые провайдеры LINQ содержат асинхронные версии этих операторов. Например, `Entity Framework` предлагает `SingleAsync`, `ContainsAsync`, `AnyAsync`, `AllAsync`, `ToDictionaryAsync` и `ToListAsync`, а также эквиваленты для других операторов, которые выполняют немедленную оценку.

Такая логика может показаться странной и бессмысленной. Так, ребенок, которого спрашивают: «Ты съел овощи?», дает бесполезный ответ: «Я съел все овощи, которые себе положил», умалчивая о том, что он вообще не клал овощи себе на тарелку. Технически это не является неправдой, но и не содержит информации, которая нужна родителю. Тем не менее операторы работают таким образом по определенной причине: они соответствуют некоторым стандартным математическим логическим операторам. `Any` — это квантор существования, который обычно записывается как обратная Е (\exists) и произносится как «существует», а `All` — квантор общности, который обычно записывается как перевернутая А (\forall) и произносится как «для всех». Математики давно пришли к соглашению касательно утверждений, которые применяют квантор общности к пустому множеству. Например, определив в как набор всех овощей, я могу утверждать, что $\forall \{v: (v \in) \wedge \text{putOnPlateByMe}(v)\} \text{ eatenByMe}(v)$. На русском языке это будет звучать так: «для каждого овоща, который я положил на свою тарелку, правдиво утверждение, что я съел этот овощ». Это утверждение считается верным, если набор овощей, который я положил на тарелку, пуст. (Возможно, математики тоже не любят овощи.) Приятно, что правильный термин для такого утверждения — *пустая истинна* (*vacuous truth*).

Конкретные элементы и поддиапазоны

Нам может понадобиться написать запрос, который выдает только один элемент. Возможно, вы ищете первый объект в списке, который соответствует определенным критериям, или, возможно, вы хотите извлечь информацию из базы данных по определенному ключу. LINQ определяет несколько операторов, которые могут это проделать, а также некоторые связанные с ними операторы для работы с поддиапазоном элементов, который может вернуть запрос.

Используйте оператор `Single`, если у вас есть запрос, который, по вашему мнению, должен выдать ровно один результат. Листинг 10.35 показывает именно такой запрос — он ищет курс по его категории и номеру, а в моих примерах данных это однозначно идентифицирует курс.

Листинг 10.35. Применение оператора `Single` к запросу

```
var q = from course in Course.Catalog
        where course.Category == "MAT" && course.Number == 101
        select course;

Course geometry = q.Single();
```

Поскольку запросы LINQ строятся путем объединения операторов, мы можем взять запрос, построенный согласно выражению запроса, и добавить еще один оператор — в данном случае оператор `Single`. Хотя большинство операторов возвращают объект, представляющий другой запрос (в данном случае `IEnumerable<T>`, так как мы используем LINQ to Objects), `Single` работает по-другому. Подобно `ToArray` и `ToList`, оператор `Single` выполняет запрос немедленно, после чего возвращает единственный объект, полученный с помощью запроса. Если запрос не может выдать ровно один объект — возможно, он не выдает ни одного или выдает два — это приводит к исключению `InvalidOperationException`. (Так как это еще один оператор, который выдает оператор немедленно, некоторые провайдеры предлагают `SingleAsync`, о чем можно прочесть во врезке «Асинхронное немедленное вычисление» на с. 556.)

Существует перегрузка оператора `Single`, которая принимает предикат. Как показывает листинг 10.36, она позволяет нам более компактно отразить логику, которую мы видим в листинге 10.35. (Как и в случае с оператором `Where`, все операторы на основе предикатов в этом разделе используют `Func<T, bool>`, а не `Predicate<T>`.)

Листинг 10.36. Оператор `Single` с предикатом

```
Course geometry = Course.Catalog.Single(  
    course => course.Category == "MAT" && course.Number == 101);
```

Оператор `Single` не прощает ошибок: если ваш запрос не возвращает ровно один элемент, он выдает исключение. Есть немного более гибкий вариант, называемый `SingleOrDefault`, который позволяет запросу вернуть либо один элемент, либо ни одного. Если запрос ничего не возвращает, то метод возвращает значение по умолчанию для типа элемента (т. е. `null`, если это ссылочный тип, `0`, если числовой, и `false`, если это тип `bool`). Несколько совпадений по-прежнему вызывают исключение. Как и в случае с `Single`, у него есть две перегрузки: одна без аргументов, для использования с источником, который, по вашему мнению, содержит не более одного объекта, и одна, которая принимает предикат лямбда.

LINQ определяет два связанных оператора, `First` и `FirstOrDefault`, каждый из которых предлагает перегрузки без аргументов или предикатов. Для последовательностей, содержащих ноль или один совпадающий элемент, они ведут себя точно так же, как `Single` и `SingleOrDefault`: возвращают элемент, если таковой имеется; в противном случае `First` выдаст исключение, а `FirstOrDefault` вернет `null` или эквивалентное значение. Однако на наличие нескольких результатов эти операторы реагируют по-разному — вместо исключения они просто возвращают первый результат, отбрасывая остальные. Это может пригодиться, если вы хотите найти самый дорогой элемент в списке — можно отсортировать запрос по убыванию цены, а затем выбрать первый результат. Листинг 10.37 использует подобную технику, чтобы выбрать самый длинный курс из моих демонстрационных данных.

Листинг 10.37. Используем `First`, чтобы выбрать самый длинный курс

```
var q = from course in Course.Catalog  
        orderby course.Duration descending  
        select course;  
Course longest = q.First();
```

Если ваш запрос не гарантирует какого-либо конкретного порядка результатов, эти операторы выберут один произвольный элемент.

Существование `First` и `FirstOrDefault` поднимает очевидный вопрос: можно ли выбрать последний элемент? Ответ — да. Для этого есть операторы `Last`

и `LastOrDefault`, и каждый из них имеет две перегрузки — одну без аргументов, а другую с предикатом.



Не используйте `First` или `FirstOrDefault`, если только вы не ожидаете, что будет несколько совпадений, и хотите обработать только одно из них. Некоторые разработчики используют их, когда ожидается только одно совпадение. Операторы, конечно, будут сработают, но `Single` и `SingleOrDefault` более точно соответствуют вашим ожиданиям. Они сообщат, когда эти ожидания не оправдались, выдав исключение в случае нескольких совпадений. Если ваш код содержит неверные предположения, лучше об этом знать, вместо того чтобы слепо продолжать.

Следующий очевидный вопрос: что, если я хочу определенный элемент, который не является ни первым, ни последним? В данном конкретном случае ответом будет использование команды LINQ, поскольку она предлагает операторы `ElementAt` и `ElementAtOrDefault`, оба из которых принимают только индекс. (Перегрузок нет.) Это обеспечивает доступ к элементам любого `IEnumerable<T>` по индексу, но будьте осторожны: если вы запрашиваете 10 000-й элемент, этим операторам может потребоваться запросить и отбросить первые 9999 элементов, чтобы добраться до нужного. Так вышло, что LINQ to Objects обнаруживает, когда исходный объект реализует `IList<T>`, и в таком случае использует индексатор для извлечения элемента напрямую, а не медленным обходным путем. Но не все реализации `IEnumerable<T>` поддерживают произвольный доступ, поэтому эти операторы могут оказаться очень медленными. В частности, даже если ваш источник реализует `IList<T>`, то после применения к нему одного или нескольких операторов LINQ выходные данные этих операторов обычно не будут поддерживать индексацию. Поэтому неразумно использовать `ElementAt` в цикле, как показано в листинге 10.38.

Листинг 10.38. Как нельзя использовать ElementAt

```
var mathsCourses = Course.Catalog.Where(c => c.Category == "MAT");
for (int i = 0; i < mathsCourses.Count(); ++i)
{
    // Никогда так не делайте!
    Course c = mathsCourses.ElementAt(i);
    Console.WriteLine(c.Title);
}
```

Несмотря на то что `Course.Catalog` является массивом, я отфильтровал его содержимое с помощью оператора `Where`, который возвращает запрос типа `IEnumerable<Course>`, не реализующий `IList<Course>`. Первая итерация сработает вполне normally — я передам `ElementAt` с индексом 0, так что получу первое совпадение, а в случае моих данных — самый первый элемент, в котором `Where` обнаружит совпадение. Но во второй проход цикла мы снова вызываем `ElementAt`. Запрос, на который ссылается `mathsCourses`, не отслеживает, куда мы попали в предыдущем цикле, — это `IEnumerable<T>`, а не `IEnumerator<T>`, — так что все начнется заново. `ElementAt` выполнит этот запрос для первого элемента, который немедленно отбросит, а затем запросит следующий элемент, который и станет возвращаемым значением. Таким образом, запрос `Where` выполнился дважды — в первый раз `ElementAt` выполнял его только для одного элемента, а затем во второй раз — для двух, поэтому он дважды обработал первый курс. В третий раз за цикл (и в последний) мы проделываем все это снова, но на этот раз `ElementAt` отбросит первые два совпадения и вернет третье, так что теперь он обратился к первому курсу уже три раза, ко второму — два, к третьему и четвертому — по одному. (Третий курс в моих данных не относится к категории МАТ, поэтому запрос `Where` будет пропущен при запросе третьего элемента.) Итак, чтобы получить три элемента, я трижды выполнил запрос `Where`, заставив его вычислить мой критерий лямбда семь раз.

По факту все еще хуже, потому что цикл `for` будет каждый раз вызывать метод `Count`, а метод `Count` с неиндексируемым источником, таким как возвращаемый `Where`, вынужден проверять всю последовательность. Это единственный способ, которым оператор `Where` может сообщить о том, сколько элементов соответствует критерию. Таким образом, этот код полностью три раза выполняет запрос, возвращаемый `Where`, в дополнение к трем частичным выполнениям, произведенным оператором `ElementAt`. Нам это сходит с рук, потому что коллекция небольшая. Но если бы у меня был массив из 1000 элементов, каждый из которых соответствовал бы фильтру, то мы бы полностью выполнили запрос `Where` 1000 раз и еще 1000 раз частично. Каждое полное выполнение вызывает предикат фильтра 1000 раз, а частичные выполнения в нашем случае будут делать это в среднем 500 раз, поэтому в итоге код выполнит фильтр 1 500 000 раз. Перебор запроса `Where` с помощью цикла `foreach` привел бы к однократному выполнению запроса. Выражение фильтра при этом выполнилось бы 1000 раз, и это привело бы к тем же результатам.

Так что будьте осторожны с `Count` и `ElementAt`. Если вы используете их в цикле, который перебирает коллекцию, для которой вы их вызываете, результирующий код будет иметь сложность $O(n^2)$.

Все операторы, которые я только что описал, возвращают из источника один элемент. Есть еще два оператора, которые также выбирают, какие элементы использовать, но могут возвращать несколько элементов: `Skip` и `Take`. Оба принимают один аргумент `int`. Как следует из названия, `Skip` отбрасывает указанное количество элементов источника, после чего возвращает все остальные. `Take` возвращает указанное количество элементов с начала последовательности, а затем отбрасывает остальные (что похоже на действие оператора `TOP` в SQL). `TakeLast` делает то же самое, за исключением того, что работает от конца. Например, вы можете использовать его, чтобы получить из источника последние 5 элементов.

Существуют и эквиваленты на основе предикатов, `SkipWhile` и `TakeWhile`. `SkipWhile` будет отбрасывать элементы из последовательности, пока не найдет соответствующий предикату, после чего вернет его и все остальные элементы до конца последовательности (независимо от того, соответствуют ли они предикату). И напротив, `TakeWhile` возвращает элементы до тех пор, пока не встретит первый элемент, который не соответствует предикату, после чего отбросит и его, и оставшуюся часть последовательности.

Хотя `Skip`, `Take`, `SkipWhile` и `TakeWhile` явно чувствительны к порядку, они не ограничены только упорядоченными типами, такими как `IOrderedEnumerable<T>`. Они также определены для `IEnumerable <T>`, что вполне разумно. Хотя конкретный порядок не гарантирован, `IEnumerable<T>` всегда выдает элементы в определенном порядке. (Единственный способ извлечь элементы из `IEnumerable<T>` — извлекать их один за другим, поэтому порядок всегда будет, пусть даже и беспорядковый. Он может быть разным при каждом переборе элементов, но для отдельного выполнения элементы будут выдаваться в некотором порядке.) Кроме того, `IOrderedEnumerable<T>` не так широко представлен за пределами LINQ, поэтому довольно часто встречаются объекты, не поддерживающие LINQ, которые производят элементы в известном порядке, но реализуют при этом только `IEnumerable<T>`. Эти операторы полезны в подобных сценариях, поэтому ограничение чуть смягчается. Немного более удивительно, что `IQueryable<T>` также поддерживает эти операции, но это согласуется с тем фактом, что многие базы данных поддерживают `TOP` (отчасти эквивалентный действию `Take`) даже для неупорядоченных запросов. Как всегда, отдельные провайдеры могут не

поддерживать отдельные операции, поэтому в сценариях, для которых нет разумной интерпретации этих операторов, они просто выдают исключение.

Агрегирование

Операторы `Sum` и `Average` складывают значения всех исходных элементов. `Sum` возвращает сумму, а `Average` возвращает сумму, поделенную на количество элементов. В поддерживающих их провайдерах LINQ они обычно доступны для коллекций элементов следующих числовых типов: `decimal`, `double`, `float`, `int` и `long`. Существуют также перегрузки, которые работают с любым типом элемента в сочетании с лямбда-выражением, которое принимает элемент и возвращает один из этих числовых типов. Это позволяет нам писать код, представленный в листинге 10.39, который работает с коллекцией объектов `Course` и вычисляет среднее арифметическое определенного значения, извлеченного из объекта, а именно продолжительности курса в часах.

Листинг 10.39. Оператор Average с проекцией

```
Console.WriteLine("Average course length in hours: {0}",  
    Course.Catalog.Average(course => course.Duration.TotalHours));
```

LINQ также определяет операторы `Min` и `Max`. Их можно применять к любому типу последовательности, хотя успех при этом не гарантирован — конкретный провайдер, который вы используете, может сообщить об ошибке, если не знает, как сравнивать используемые вами типы. Например, LINQ to Objects требует, чтобы объекты в последовательности реализовывали `IComparable`.

И `Min`, и `Max` имеют перегрузки, которые принимают лямбду, которая получает из исходного элемента значение для использования. Листинг 10.40 таким образом находит дату публикации самого последнего курса.

Листинг 10.40. Max с проекцией

```
DateTime m = mathsCourses.Max(c => c.PublicationDate);
```

Обратите внимание, что пример не возвращает курс с самой последней датой публикации. Результатом является дата публикации этого курса. Если вы хотите получить объект, для которого конкретное свойство имеет максимальное значение, вы можете использовать оператор `OrderByDescending`, после которого следует `First`. Тем не менее в данном случае вы просите отсортировать входные данные целиком, так что это может оказаться несколько

неэффективным подходом, поскольку потенциально приведет к большему объему работы.

LINQ to Objects определяет специализированные перегрузки `Min` и `Max` для последовательностей, которые возвращают те же числовые типы, с которыми имеют дело `Sum` и `Average` (т. е. `decimal`, `double`, `float`, `int` и `long`). Он также определяет аналогичные специальные варианты, принимающие лямбду. Эти перегрузки предназначены для повышения производительности за счет исключения упаковки. Вариант общего назначения опирается на `IComparable`, так что получение ссылки на значение типа интерфейса всегда включает в себя упаковку этого значения. Для больших коллекций упаковка каждого отдельного значения добавит значительный объем работы сборщику мусора.



Microsoft предоставляет пакет NuGet под названием `System.Interactive`, который содержит различные дополнительные операторы в стиле LINQ². Он включает в себя `MaxBy`, который является более прямым (и скорее всего, более эффективным) способом достижения того же результата. Если вам и этого мало, существует проект `MoreLINQ`, не относящийся к Microsoft.

LINQ определяет оператор под названием `Aggregate`, который обобщает шаблон, совместно используемый `Min`, `Max`, `Sum` и `Average`. Согласно ему, в результате процесса, включающего в себя оценку каждого исходного элемента, выдается единственный результат. Можно реализовать все четыре этих оператора в форме `Aggregate`. В листинге 10.41 оператор `Sum` используется для расчета общей продолжительности всех курсов, после чего показано, как для выполнения точно такого же вычисления использовать оператор `Aggregate`.

Листинг 10.41. `Sum` и эквивалент с использованием `Aggregate`

```
double t1 = Course.Catalog.Sum(course => course.Duration.TotalHours);
double t2 = Course.Catalog.Aggregate(
    0.0, (hours, course) => hours + course.Duration.TotalHours);
```

² Здесь требуется пояснение. Это часть проекта «Реактивные расширения», описанного в главе 11. Пакет `System.Reactive` определяет стандартные операторы LINQ для интерфейса под названием `IObservable<T>`, но также добавляет множество полезных нестандартных операторов. `System.Interactive` содержит все эти дополнительные операторы для `IEnumerable<T>`, и его имя выглядит как своего рода зеркальное отражение `System.Reactive`.

Агрегация работает путем создания значения, которым представлено все, что мы знаем обо всех проверяемых элементах. Это значение называется **аккумулятором**. Тип, который мы используем, зависит от сведений, которые мы хотим аккумулировать. В моем случае я просто складываю все числа, так что я использую `double` (потому что свойство `TotalHours` типа `TimeSpan` тоже `double`).

Изначально у нас нет никаких сведений, потому что мы еще не рассмотрели ни одного элемента. Нам нужно значение аккумулятора, представляющее эту начальную точку, поэтому первый аргумент оператора `Aggregate` — это `seed`, начальное значение для аккумулятора. В листинге 10.41 аккумулятор является просто промежуточным итогом, поэтому начальное значение равно `0,0`.

Второй аргумент — это лямбда, которая описывает, как обновить аккумулятор, чтобы добавить в него информацию от одного элемента. Так как в данном случае моя цель — подсчитать общее время, я просто добавляю к промежуточному итогу продолжительность текущего курса.

Как только `Aggregate` просмотрел каждый элемент, эта конкретная перегрузка непосредственно возвращает аккумулятор. В данном случае это окажется общее количество часов во всех курсах. Аккумулятор не обязан использовать сложение. Мы можем реализовать `Max`, используя тот же процесс с другой стратегией аккумуляции. Вместо того чтобы хранить промежуточный итог, мы представляем все, что знаем о данных, самым большим из встреченных значений. Листинг 10.42 показывает примерный эквивалент листинга 10.40. (Он не точно такой же, так как листинг 10.42 не пытается обнаружить пустой источник. `Max` выдаст исключение, если источник пуст, но данный пример просто вернет дату `0/0/0000`.)

Листинг 10.42. Реализация `Max` через `Aggregate`

```
DateTime m = mathsCourses.Aggregate (
    new DateTime(),
    (date, c) => date > c.PublicationDate ? date : c.PublicationDate);
```

Из этого видно, что оператор `Aggregate` не вкладывает какой-либо единственный смысл в значение, которое сохраняет сведения. Его использование зависит от того, что вы делаете. Некоторые операции требуют более сложного аккумулятора. Листинг 10.43 с помощью `Aggregate` вычисляет среднюю продолжительность курса.

Листинг 10.43. Реализация Average через Aggregate

```
double average = Course.Catalog.Aggregate(  
    new { TotalHours = 0.0, Count = 0 },  
    (totals, course) => new  
    {  
        TotalHours = totals.TotalHours + course.Duration.TotalHours,  
        Count = totals.Count + 1  
    },  
    totals => totals.Count >= 0  
        ? totals.TotalHours / totals.Count  
        : throw new InvalidOperationException("Sequence was empty"));
```



Листинг 10.43 опирается на тот факт, что, если два отдельных метода в одном компоненте создают экземпляры двух структурно идентичных анонимных типов, компилятор генерирует один тип, который используется в обоих случаях. Начальное значение — экземпляр анонимного типа, состоящий из `double` с именем `TotalHours` и `int` с именем `Count`. Лямбда-аккумулятор также возвращает экземпляр анонимного типа с теми же именами и типами членов в том же порядке. Компилятор C# считает, что это будет один и тот же тип, и это важно, так как `Aggregate` требует, чтобы лямбда принимала и возвращала экземпляр типа аккумулятора.

Чтобы вычислить среднюю продолжительность, нужно знать две вещи: общую продолжительность и количество элементов. Следовательно, в этом примере мой аккумулятор использует тип, который может содержать два значения: одно для итоговой суммы и одно для количества элементов. Я использовал анонимный тип, потому что, как уже упоминалось, в LINQ это зачастую единственный вариант, а я хочу показать наиболее общий случай. Тем не менее стоит отметить, что в этом конкретном случае кортеж был бы более предпочтительным. Он будет работать, потому что это LINQ to Objects, и, поскольку облегченные кортежи являются значимыми типами, тогда как анонимные типы — ссылочными, кортеж уменьшит количество выделяемых объектов.

В листинге 10.43 используется другая перегрузка, нежели в предыдущем примере. Она дополнительно принимает лямбду, которая используется для извлечения из аккумулятора возвращаемого значения — в данном примере аккумулятор накапливает информацию, необходимую для получения результата, но сам при этом не является результатом.

Конечно, если все, что вам нужно, — это вычислить сумму, максимальное или среднее значение, вы не будете использовать `Aggregate`. Вместо этого

вы примените специализированные операторы, предназначенные для выполнения этих задач. Они не только проще, но и эффективнее. (Например, провайдер LINQ для базы данных может генерировать запрос, который использует встроенные функции базы данных для вычисления минимального или максимального значения.) Используя доступные примеры, я просто хотел продемонстрировать гибкость. Но теперь, когда я это сделал, в листинге 10.44 я покажу особенно лаконичное использование `Aggregate`, которое не соответствует ни одному другому встроенному оператору. Пример принимает коллекцию прямоугольников и возвращает ограничивающий прямоугольник, который содержит все прямоугольники.

Листинг 10.44. Агрегирование ограничивающих прямоугольников

```
public static Rect GetBounds(IEnumerable<Rect> rects) =>
    rects.Aggregate(Rect.Union);
```

В этом примере использована структура `Rect` из пространства имен `System.Windows`. Это очень простая структура данных из состава WPF, которая содержит только четыре числа — `X`, `Y`, `Width` и `Height`, так что если хотите, вы можете использовать ее в приложениях, не использующих WPF. В листинге 10.44 используется статический метод `Union` типа `Rect`, который принимает два аргумента `Rect` и возвращает один `Rect`, являющийся ограничивающим прямоугольником двух входных прямоугольников (т. е. наименьший прямоугольник, который их содержит)³.

В данном случае я использую простейшую перегрузку `Aggregate`. Она делает то же самое, что и использованная в листинге 10.41, но не требует представления начального значения, просто используя первый элемент в списке. Листинг 10.45 эквивалентен листингу 10.44, но шаги в нем отражены более явно. Я предоставил первый `Rect` последовательности в качестве явного начального значения, используя `Skip` для агрегирования всего, кроме этого первого элемента. Я также написал лямбду для вызова метода вместо передачи самого метода. Если вы используете лямбду такого типа, которая всего лишь передает свои аргументы прямо в существующий метод с помощью LINQ to Objects, вы можете просто передать имя метода и он будет вызывать

³ Если вы так делаете, то будьте осторожны и не перепутайте его с другим типом WPF `Rectangle`. Он намного сложнее и поддерживает анимацию, стили, макет, пользовательский ввод, привязку данных и различные другие функции WPF. Не пытайтесь использовать `Rectangle` вне приложения WPF.

целевой метод напрямую, а не через вашу лямбду. (Вы не можете сделать это с провайдерами на основе выражений, так как им нужна лямбда.)

Непосредственное использование метода более лаконично и немного более эффективно, но оно же делает код менее ясным, поэтому в листинге 10.45 я объясняю все детально.

Листинг 10.45. Более многословная и более ясная агрегация ограничивающих прямоугольников

```
public static Rect GetBounds(IEnumerable<Rect> rects)
{
    IEnumerable<Rect> theRest = rects.Skip(1);
    return theRest.Aggregate(rects.First(), (r1, r2) =>
        Rect.Union(r1, r2));
}
```

Эти два примера работают одинаково. Они начинают с первого прямоугольника в качестве начального значения. Для следующего элемента в списке `Aggregate` будет вызывать `Rect.Union`, передавая начальное значение и второй прямоугольник. Результат — ограничивающий прямоугольник первых двух прямоугольников — становится новым значением аккумулятора. Затем он передается в `Union` вместе с третьим прямоугольником и т. д. В листинге 10.46 показано, каким будет результат этой операции `Aggregate`, если она выполняется для набора из четырех значений `Rect`. (Четыре значения представлены как `r1`, `r2`, `r3` и `r4`. Чтобы передать их в `Aggregate`, они должны находиться внутри коллекции, например в массиве.)

Листинг 10.46. Результат `Aggregate`

```
Rect bounds = Rect.Union(Rect.Union(Rect.Union(r1, r2), r3), r4);
```

Как я упоминал ранее, `Aggregate` — это имя операции, иногда называемой `reduce` (обобщение). Иногда ее называют сверткой списка. Имя `Aggregate` используется в LINQ по той же причине, по которой оператор проекции называется `Select`, а не `map` (более распространенное имя в функциональных языках программирования): на терминологию LINQ большее влияние оказал язык SQL, нежели функциональные языки программирования.

Операции над множествами

LINQ определяет три оператора, которые для объединения двух источников используют некоторые распространенные операции над множествами.

`Intersect` возвращает результат, который содержит только те элементы, которые были в обоих входных источниках. `Except` включает в результат только те элементы из первого источника, которых не было во втором. Результат `Union` содержит элементы, которые были в одном (или в обоих) из входных источников⁴.

Хотя LINQ определяет эти операции над множествами, большинство типов источников LINQ напрямую не соответствуют абстракции множества. В математике любой конкретный элемент либо принадлежит множеству, либо нет, без какой-либо внутренней концепции порядка или количества повторных вхождений. `IEnumerable<T>` работает не так. Это последовательность элементов, которая может содержать дубликаты и то же самое относится к `IQueryable<T>`. Это не всегда проблема, потому что некоторые коллекции никогда не оказываются в ситуации, когда они содержат дубликаты, а в некоторых случаях и само наличие дубликатов не представляет проблемы. Однако иногда бывает нужно удалить дубликаты из коллекции. Для этого LINQ определяет оператор `Distinct`, который это и делает. Листинг 10.47 содержит запрос, который извлекает имена категорий всех курсов, а затем передает их в оператор `Distinct`, чтобы убедиться, что каждое имя категории используется только единожды.

Листинг 10.47. Удаление дубликатов с помощью `Distinct`

```
var categories = Course.Catalog.Select(c => c.Category).Distinct();
```

Все эти операторы множеств доступны в двух формах, потому что вы можете по желанию передать в любой из них `IEqualityComparer<T>`. Это позволяет вам настроить способ, с помощью которого операторы определяют, являются ли два элемента одинаковыми.

Сохраняющие порядок операции на всей последовательности

LINQ определяет операторы, выходные данные которых включают в себя каждый элемент из источника и которые при этом сохраняют или изменяют порядок. Не все коллекции обязательно упорядочены, поэтому эти операторы поддерживаются не всегда. Однако LINQ to Objects поддерживает их в полной мере. Самым простым является `Reverse`, который обращает порядок элементов.

⁴ Это не связано с методом `Rect.Union`, использованным в предыдущем примере.

Оператор `Concat` объединяет две последовательности. Он возвращает последовательность, которая выдает все элементы из первой последовательности (в том порядке, в котором эта последовательность их возвращает), за которой следуют все элементы из второй последовательности (опять же, с сохранением порядка). В тех случаях, когда нужно добавить только один элемент в конец первой последовательности, вы можете использовать `Append`. Имеется также `Prepend`, который добавляет один элемент в начале. Оператор `Repeat` по сути объединяет указанное количество копий источника.

Оператор `DefaultIfEmpty` возвращает все элементы из своего источника. Однако если источник пуст, он возвращает единственный элемент, который имеет нулевое значение по умолчанию для его типа элемента.

Оператор `Zip` также объединяет две последовательности, но вместо того, чтобы возвращать одну за другой, он работает с парами элементов. Таким образом, первый элемент, который он возвращает, будет основан на первых элементах из первой и из второй последовательности. Второй элемент в такой «застегнутой» последовательности (`zip` – застегивать, англ.) будет основан на вторых элементах каждой последовательности и т. д. Название `Zip` навеяно тем, как молния в предмете одежды идеально объединяет две части чего-либо. (Это не точная аналогия. Когда застежка-молния застегивается, зубцы двух половинок чередуются. Но оператор `Zip` не чередует свои входные данные, подобно зубцам физической молнии. Он объединяет предметы из двух источников в пары.)

Поскольку `Zip` работает с парами элементов, вам нужно указать, как вы хотите их объединять. Он принимает лямбду с двумя аргументами и передает пары элементов из двух источников в качестве этих аргументов, выдавая то, что ваша лямбда возвращает в качестве выходных элементов. В листинге 10.48 используется селектор, который объединяет каждую пару элементов с использованием конкатенации строк.

Листинг 10.48. Объединение списков с помощью `Zip`

```
string[] firstNames = { "Carmel", "Ed", "Arthur", "Arthur" };
string[] lastNames = { "Eve", "Freeman", "Dent", "Pewty" };
IEnumerable<string> fullNames = firstNames.Zip(lastNames,
    (first, last) => first + " " + last);
foreach (string name in fullNames)
{
    Console.WriteLine(name);
}
```

Два списка, которые объединяет этот пример, содержат соответственно имена и фамилии. Вывод выглядит так:

Carmel Eve
Ed Freeman
Arthur Dent
Arthur Pewty



.NET Core добавляет перегрузку `Zip`, которая не требует лямбды. Он просто возвращает последовательность кортежей.

Если входные источники содержат различное количество элементов, `Zip` остановится, как только достигнет конца кратчайшей коллекции, и не будет пытаться получить какие-либо дополнительные элементы из более длинной. Он не рассматривает несоответствие длины как ошибку.

Оператор `SequenceEqual` имеет сходство с `Zip` в том смысле, что работает с парами элементов, которые находятся в одной и той же позиции в двух последовательностях. Но вместо того, чтобы передавать их в лямбду для объединения, `SequenceEqual` просто сравнивает каждую пару. Если процесс сравнения обнаружит, что два источника содержат одинаковое количество элементов и что в каждой паре эти два элемента равны, то он возвращает `true`. Если источники имеют разную длину или хотя бы одна пара элементов не совпадает, возвращается `false`. Оператор `SequenceEqual` имеет две перегрузки, одна из которых принимает только список, с которым сравнивается источник, а другая еще и `IEqualityComparer<T>`, который сообщает, что именно вы подразумеваете под равенством.

Группировка

Иногда все элементы, которые имеют что-то общее, нужно обработать как группу. Листинг 10.49 использует запрос для группировки курсов по категориям, выводя заголовок для каждой категории перед тем, как перечислить все курсы этой категории.

Листинг 10.49. Выражения запроса группировки

```
var subjectGroups = from course in Course.Catalog
                     group course by course.Category;
```

```
foreach (var group in subjectGroups)
{
    Console.WriteLine("Category: " + group.Key);
    Console.WriteLine();

    foreach (var course in group)
    {
        Console.WriteLine(course.Title);
    }
    Console.WriteLine();
}
```

group принимает выражение, определяющее членство в группе, — в данном случае любые курсы, свойства Category которых возвращают одно и то же значение, будут считаться принадлежащими к одной группе. Выражение group создает коллекцию, в которой каждый элемент реализует тип, представляющий группу. Поскольку я использую LINQ to Objects и группирую по строке категории, тип переменной subjectGroup в листинге 10.49 будет `IEnumerable<IGrouping<string, Course>>`. Такой пример создает три объекта группы, изображенных на рис. 10.1.

Каждый из элементов `IGrouping<string, Course>` имеет свойство Key, и поскольку запрос группирует элементы по свойству Category курса, каждый ключ содержит строковое значение, полученное из этого свойства. В данных листинга 10.17 имеются три разные категории: MAT, BIO и CSE, поэтому они становятся значениями Key для трех групп.

Интерфейс `IGrouping<TKey, TItem>` наследуется от `IEnumerable<TItem>`, следовательно, каждый объект группы можно перебрать, чтобы получить содержащиеся в нем элементы. Так что в листинге 10.49 внешний цикл `foreach` выполняет итерации по трем возвращаемым группам, а затем внутренний цикл `foreach` выполняет итерации по объектам `Course` в каждой из групп.

Выражение запроса превращается в код из листинга 10.50.

Листинг 10.50. Разворачивание простого группирующего запроса

```
var subjectGroups = Course.Catalog.GroupBy(course => course.Category);
```

По части группировки выражения запроса предлагают кое-какие вариации. Небольшой модификацией исходного запроса мы можем расставить элементы в каждой группе по-другому, чем это было сделано изначально.

В листинге 10.51 я изменил выражение сразу после ключевого слова `group` с `course` на `course.Title`.

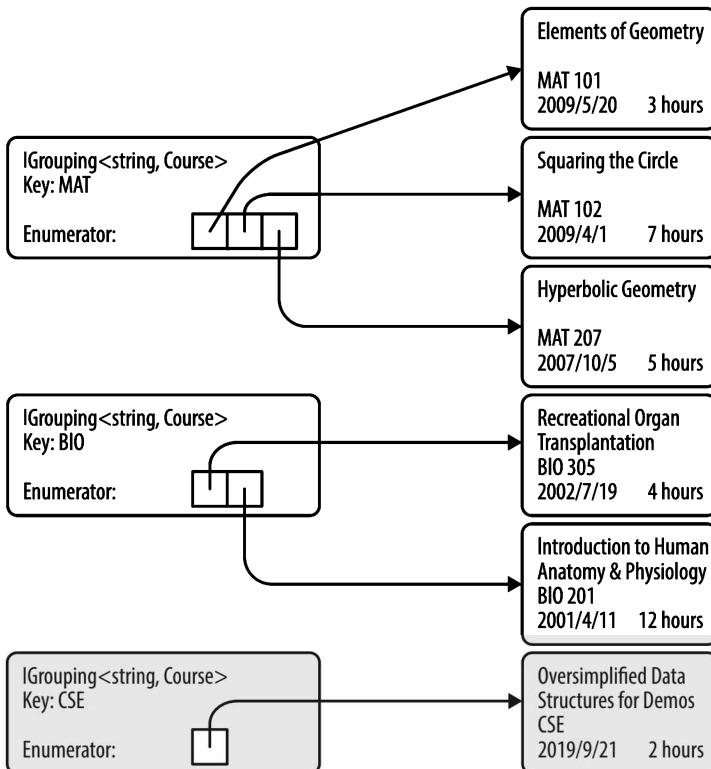


Рис. 10.1. Результат выполнения группирующего запроса

Листинг 10.51. Группирующий запрос с помощью проекции элемента

```
var subjectGroups = from course in Course.Catalog
                    group course by course.Category;
```

Здесь все еще присутствует группирующее выражение, `course.Category`, так что в результате, как и раньше, получаются три группы. Но тип поменялся на `IGrouping<string, string>`. Если перебрать содержимое одной из групп, обнаружится, что каждая группа содержит последовательность строк с названиями курсов. Как показано в листинге 10.52, компилятор разворачивает этот запрос в иную перегрузку оператора `GroupBy`.

Листинг 10.52. Расширение группирующего запроса с проекцией элемента

```
var subjectGroups = Course.Catalog
    .GroupBy(course => course.Category, course => course.Title);
```

Выражения запроса в качестве последнего выражения должны содержать либо `select`, либо `group`. Но если запрос содержит выражение `group`, оно не обязано быть последним. В листинге 10.51 я изменил то, как запрос представляет каждый элемент в группе (т. е. поля справа на рис. 10.1), но я также свободно могу настраивать объекты, представляющие каждую группу (элементы слева). По умолчанию я получаю объекты типа `IGrouping< TKey, TItem >` (или эквивалент, зависящий от провайдера LINQ), но это можно изменить. Листинг 10.53 использует необязательное ключевое слово `into` в выражении `group`. Оно вводит новую переменную диапазона, которая перебирает объекты группы и которую я могу продолжать использовать в оставшейся части запроса. Я мог бы продолжить запрос другими типами выражений, такими как `orderby` или `where`, но в данном случае решил использовать выражение `select`.

Листинг 10.53. Группирующий запрос с проекцией группы

```
var subjectGroups =
    from course in Course.Catalog
    group course by course.Category into category
    select $"Category '{category.Key}' contains {category.Count()} courses";
```

Результатом этого запроса является `IEnumerable<string>`, и, если вы отобразите все строки, которые он выдает, вы получите вот что:

```
Category 'MAT' contains 3 courses
Category 'BIO' contains 2 courses
Category 'CSE' contains 1 courses
```

Как показано в листинге 10.54, запрос расширяется до вызова той же переопределки `GroupBy`, которую использует листинг 10.50, после чего для заключительного выражения используется обычный оператор `Select`.

Листинг 10.54. Развернутый группирующий запрос с проекцией группы

```
IEnumerable<string> subjectGroups = Course.Catalog
    .GroupBy(course => course.Category)
    .Select(category =>
        $"Category '{category.Key}' contains {category.Count()} courses");
```

LINQ to Objects определяет еще несколько перегрузок для оператора `GroupBy`, которые недоступны из синтаксиса запроса. В листинге 10.55 показана перегрузка, которая обеспечивает чуть более непосредственный эквивалент листинга 10.53.

Листинг 10.55. `GroupBy` с проекциями ключа и группы

```
IEnumerable<string> subjectGroups = Course.Catalog.GroupBy(
    course => course.Category,
    (category, courses) =>
        $"Category '{category}' contains {courses.Count()} courses");
```

Данная перегрузка принимает две лямбды. Первая — это выражение, согласно которому элементы группируются. Вторая используется для создания каждого объекта группы. В отличие от предыдущих примеров, здесь не используется интерфейс `IGrouping< TKey, TItem >`. Вместо этого конечная лямбда получает ключ в качестве первого аргумента, а затем набор элементов в группе в качестве второго. Это та же информация, которую инкапсулирует `IGrouping< TKey, TItem >`, но поскольку данная форма оператора может передавать их как отдельные аргументы, это устраниет необходимость в объектах для представления групп.

Имеется еще одна версия этого оператора, показанная в листинге 10.56. Он сочетает в себе функциональность всех остальных.

Листинг 10.56. Оператор `GroupBy` с проекциями ключа, элемента и группы

```
IEnumerable<string> subjectGroups = Course.Catalog.GroupBy(
    course => course.Category,
    course => course.Title,
    (category, titles) =>
        $"Category '{category}' contains {titles.Count()} courses: " +
        string.Join(", ", titles));
```

Эта перегрузка принимает три лямбды. Первая — это выражение, согласно которому элементы группируются. Второй определяет, как именно представлены отдельные элементы в группе — на этот раз я решил извлечь название курса. Третья лямбда используется для создания каждого группового объекта, и, как в листинге 10.55, этой последней лямбде в качестве одного аргумента передается ключ, а в качестве другого — элементы группы, преобразованные второй лямбдой. Таким образом, вместо оригинальных элементов `Course` вторым аргументом будет `IEnumerable<string>`, содержащий

названия курсов, потому что это именно то, что запрашивала вторая лямбда в этом примере. Результат этого оператора `GroupBy` тоже представляет собой набор строк, но теперь он выглядит так:

```
Category 'MAT' contains 3 courses: Elements of Geometry, Squaring the Circle, Hyperbolic Geometry
Category 'BIO' contains 2 courses: Recreational Organ Transplantation, Introduction to Human Anatomy and Physiology
Category 'CSE' contains 1 courses: Oversimplified Data Structures for Demos
```

Я показал четыре версии оператора `GroupBy`. Все четыре принимают лямбду, которая определяет ключ для группировки, и простейшая перегрузка не требует больше ничего. Остальные позволяют вам управлять представлением отдельных элементов в группе, или представлением каждой группы, или и тем и другим. Существует еще четыре версии этого оператора. Они предоставляют все те же возможности, что и четыре предыдущие, но так же принимают `IEqualityComparer<T>`, что позволяет настроить логику определения того, что два ключа одинаковы.

Иногда может быть полезна группировка по нескольким значениям. Предположим, вы хотите сгруппировать курсы по категориям и годам публикации. Можно объединить операторы, группируя сначала по категориям, а затем по годам внутри категории (или наоборот). Но такой уровень вложенности может быть избыточным — вместо групп из групп вам может понадобиться сгруппировать курсы по каждой уникальной комбинации категории и года публикации. Этого можно добиться, просто поместив оба значения в ключ, и вы можете использовать для этого анонимный тип, как показано в листинге 10.57.

Листинг 10.57. Составной ключ группы

```
var bySubjectAndYear =
    from course in Course.Catalog
    group course by new { course.Category, course.PublicationDate.Year };
foreach (var group in bySubjectAndYear)
{
    Console.WriteLine($"{group.Key.Category} ({group.Key.Year})");
    foreach (var course in group)
    {
        Console.WriteLine(course.Title);
    }
}
```

Пример использует тот факт, что анонимные типы реализуют `Equals` и `GetHashCode` за нас. Это работает для всех форм оператора `GroupBy`. С провайдерами LINQ, которые не рассматривают свои лямбды как выражения (например, `LINQ to Objects`), вы можете вместо этого использовать кортеж, что будет несколько более лаконичным подходом, но при этом будет иметь тот же эффект.

Есть еще один группирующий свои выходные данные оператор, `GroupJoin`, и он делает это как часть операции объединения. Но сначала мы рассмотрим более простые объединения.

Объединение данных

LINQ определяет оператор `Join`, который позволяет запросу из одного источника использовать связанные данные из какого-либо другого источника, так же как запрос базы данных может объединять информацию из одной таблицы с данными в другой. Предположим, что в нашем приложении имеется список, содержащий данные о том, на какие курсы подписались те или иные студенты. Если вы храните эту информацию в файле, то не захотите копировать полную информацию о курсе или студенте в каждую строку — достаточно было бы минимальной информации, нужной для идентификации студента и конкретного курса. В моем примере данных курсы однозначно идентифицируются по комбинации категории и номера. Следовательно, чтобы записать, кто на что подписался, нам понадобятся записи, содержащие три фрагмента информации: категория курса, номер курса и что-то для идентификации студента. Класс в листинге 10.58 показывает, как мы можем представить такую запись в памяти.

Листинг 10.58. Класс, связывающий студента с курсом

```
public class CourseChoice
{
    public int StudentId { get; set; }

    public string Category { get; set; }

    public int Number { get; set; }
}
```

После того как наше приложение загрузит эту информацию в память, нам может понадобиться доступ к объектам курса, а не только к информации,

идентифицирующей курс. Мы можем добиться этого с помощью выражения `join`, как показано в листинге 10.59 (который, используя класс `CourseChoice`, также предоставляет кое-какие дополнительные демонстрационные данные, чтобы у запроса было с чем работать).

Листинг 10.59. Запрос с выражением `join`

```
CourseChoice[] choices =
{
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 101 },
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 102 },
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 207 },
    new CourseChoice { StudentId = 2, Category = "MAT", Number = 101 },
    new CourseChoice { StudentId = 2, Category = "BIO", Number = 201 },
};

var studentsAndCourses = from choice in choices
                        join course in Course.Catalog
                        on new { choice.Category, choice.Number }
                        equals new { course.Category, course.Number }
                        select new { choice.StudentId, Course = course };

foreach (var item in studentsAndCourses)
{
    Console.WriteLine(
        $"Student {item.StudentId} will attend {item.Course.Title}");
}
```

Пример отображает одну строку для каждой записи в массиве `choices`. Он показывает заголовок каждого курса, так как, хотя тот и не был доступен во входной коллекции, выражение `join` нашло соответствующий элемент в каталоге курсов. В листинге 10.60 показано, как компилятор транслирует запрос из листинга 10.59.

Листинг 10.60. Использование оператора `Join` напрямую

```
var studentsAndCourses = choices.Join(
    Course.Catalog,
    choice => new { choice.Category, choice.Number },
    course => new { course.Category, course.Number },
    (choice, course) => new { choice.StudentId, Course = course });
```

Задача оператора `Join` — найти во второй последовательности элемент, который соответствует элементу в первой. Соответствие определяется

первыми двумя лямбдами; элементы из двух источников будут считаться соответствующими друг другу, если значения, возвращаемые этими двумя лямбдами, равны. Пример использует анонимный тип и зависит от того факта, что два структурно идентичных экземпляра с анонимным типом в одной сборке получают один и тот же тип. Другими словами, обе эти лямбды выдают объекты одного типа. Компилятор генерирует метод `Equals` для любого анонимного типа, который по очереди сравнивает каждый член, поэтому суть этого кода в том, что две строки считаются соответствующими друг другу, если их свойства `Category` и `Number` равны. (Повторюсь, в случае провайдеров на основе `IQueryable<T>` следует использовать анонимные типы вместо кортежей, потому что эти лямбды будут превращены в деревья выражений. Но, поскольку в данном примере используется провайдер не на основе выражений, можно немного упростить код, использовав кортежи.)

Я настроил этот пример так, чтобы совпадение было только одно, но что произойдет, если категория и номер курса по какой-то причине не будут однозначно определять курс? Если для какой-либо одной входной строки имеется несколько совпадений, оператор `Join` выдаст один выходной элемент для каждого соответствия. Поэтому в таком случае мы получим больше выходных элементов, чем было записей в массиве `choices`. И наоборот, если у элемента в первом источнике нет соответствующего элемента во второй коллекции, `Join` не будет производить для такого элемента никакого вывода — он фактически его проигнорирует.

LINQ предлагает альтернативный тип объединения, который обрабатывает входные строки с несколькими соответствующими строками или вообще без них иначе, чем оператор `Join`. Листинг 10.61 показывает модифицированное выражение запроса. (Разница заключается в добавлении `into courses` в конце выражения `join`, а последнее выражение `select` ссылается на него вместо переменной диапазона `course`.) Это производит выходные данные в другой форме, поэтому я дополнительно изменил код, который выводит результаты.

Листинг 10.61. Групповое объединение

```
var studentsAndCourses =
    from choice in choices
    join course in Course.Catalog
        on new { choice.Category, choice.Number }
        equals new { course.Category, course.Number }
    into courses
    select new { choice.StudentId, Courses = courses };
```

```
foreach (var item in studentsAndCourses)
{
    Console.WriteLine($"Student {item.StudentId} will attend " +
        string.Join(", ", item.Courses.Select(course => course.Title)));
}
```

Как показано в листинге 10.62, это заставляет компилятор использовать вызов оператора `GroupJoin` вместо `Join`.

Листинг 10.62. Оператор `GroupJoin`

```
var studentsAndCourses = choices.GroupJoin(
    Course.Catalog,
    choice => new { choice.Category, choice.Number },
    course => new { course.Category, course.Number },
    (choice, courses) => new { choice.StudentId, Courses = courses });
```

Эта форма объединения выдает один результат для каждого элемента входной коллекции путем вызова последней лямбды. Ее первым аргументом выступает элемент ввода, а вторым — коллекция всех соответствующих объектов из второй коллекции. (Сравните это с `Join`, который вызывает свою последнюю лямбду по разу для каждого совпадения, передавая соответствующие элементы по одному.) Это обеспечивает способ представления элемента ввода, который не имеет соответствующих элементов во второй коллекции: оператор может просто передать пустую коллекцию.

И `Join`, и `GroupJoin` имеют перегрузки, которые принимают `IEqualityComparer<T>`, так что вы можете определить собственный критерий равенства значений, возвращаемых первыми двумя лямбдами.

Преобразование

Иногда вам нужно будет преобразовать запрос одного типа в другой. Например, вы получили коллекцию, в которой аргумент типа задает некоторый базовый тип (например, `object`), но у вас есть веские основания полагать, что коллекция на самом деле содержит элементы более конкретного типа (например, `Course`). При работе с отдельными объектами вы можете просто использовать синтаксис приведения C# для преобразования ссылки в тип, с которым, по вашему мнению, вы имеете дело. К сожалению, это не работает для таких типов, как `IEnumerable<T>` или `IQueryable<T>`.

Хотя ковариантность означает, что `IEnumerable<Course>` неявно конвертируется в `IEnumerable<object>`, вы не можете конвертировать в обрат-

ном направлении даже с помощью явного нисходящего преобразования. Если у вас есть ссылка типа `IEnumerable<object>`, то попытка привести ее к `IEnumerable<Course>` будет успешной, только если объект реализует `IEnumerable<Course>`. Вполне возможно получить в итоге последовательность, которая целиком состоит из объектов `Course`, но не реализует `IEnumerable<Course>`. Листинг 10.63 создает именно такую последовательность, и он выдает исключение при попытке приведения к `IEnumerable<Course>`.

Листинг 10.63. Как нельзя приводить последовательность

```
IEnumerable<object> sequence = Course.Catalog.Select(c => (object) c);
var courseSequence = (IEnumerable<Course>) sequence;
// InvalidCastException
```

Это, конечно, показательный пример. Я вызвал создание `IEnumerable<object>` приведением к `object` типа возвращаемого значения лямбды. Тем не менее достаточно легко по-настоящему оказаться в этой ситуации, только в несколько более сложных обстоятельствах. К счастью, есть простое решение. Вы можете использовать оператор `Cast<T>`, показанный в листинге 10.64.

Листинг 10.64. Приведение последовательности

```
var courseSequence = sequence.Cast<Course>();
```

Он возвращает запрос, который выдает каждый элемент из своего источника по порядку и при этом приводит его к указанному целевому типу. Это означает, что, хотя первоначальный `Cast<T>` мог сработать, вполне возможно, что вы получите `InvalidOperationException` через некоторое время, когда попытаетесь извлечь значения из последовательности. Ведь единственный способ, которым оператор `Cast<T>` может проверить, что заданная вами последовательность действительно всегда выдает значения типа `T`, — это извлечь все эти значения и попытаться их привести. Он не может получить всю последовательность заранее, потому что она может быть бесконечной. Если первый миллиард произведенных вашей последовательностью элементов будет правильного типа, но после этого вы вернете один из несовместимых типов, единственный способ, которым `Cast<T>` может это обнаружить, — это попытаться привести элементы по одному.

`LINQ to Objects` определяет оператор `AsEnumerable<T>`. Он просто возвращает источник без изменений, т. е. ничего не делает. Он призван принудить вас использовать `LINQ to Objects`, даже если вы имеете дело с чем-то, что можно обработать другим провайдером `LINQ`. Предположим, у вас есть что-то, что

реализует `IQueryable<T>`. Этот интерфейс наследуется от `IEnumerable <T>`, но методы расширения, которые работают с `IQueryable<T>`, будут иметь приоритет над `LINQ to Objects`. Если вы намереваетесь выполнить определенный запрос к базе данных, а затем обрабатывать результаты на стороне клиента с помощью `LINQ to Objects`, то можете использовать `AsEnumerable<T>`, чтобы провести черту, которая как бы скажет: «Здесь мы передаем все на сторону клиента».



`Cast<T>` и `OfType<T>` похожи, и разработчики иногда используют один, когда следовало бы использовать другой (обычно потому, что не подозревают о его существовании). `OfType<T>` делает почти то же самое, что и `Cast<T>`, но вместо вызова исключения он молча отфильтровывает любые элементы неправильного типа. Если вы знаете об элементах неправильного типа и хотите их игнорировать, используйте `OfType<T>`. Если вы вообще не ожидаете наличия элементов неправильного типа, используйте `Cast<T>`, потому что если вы ошибаетесь, он сообщит вам об этом с помощью исключения, уменьшив риск того, что потенциальная ошибка останется скрытой.

В противовес этому существует также `AsQueryable<T>`. Он предназначен для использования в сценариях, где есть переменная статического типа `IEnumerable<T>`, которая, по вашему мнению, может содержать ссылку на объект, также реализующий `IQueryable<T>`. При этом вы хотите убедиться, что любые создаваемые вами запросы используют его вместо `LINQ to Objects`. Если вы используете этот оператор в источнике, не реализующем `IQueryable<T>`, он вернет обертку, которая реализует `IQueryable<T>`, но подспудно использует `LINQ to Objects`.

Еще одним оператором для выбора другой разновидности `LINQ` является `AsParallel`. Он возвращает `ParallelQuery<T>`, что позволяет создавать запросы, которые будут выполняться в `Parallel LINQ`. Это провайдер `LINQ`, который способен выполнять определенные операции параллельно, чтобы повысить производительность при наличии нескольких ядер ЦП.

Есть ряд операторов, которые преобразуют запрос в другие типы, но вместо создания нового запроса, связанного с предыдущим, выполняют его немедленно. `ToArray`, `ToList` и `ToHashSet` возвращают соответственно массив, список или хеш-множество, которые содержат полные результаты выполнения входного запроса. `ToDictionary` и `ToLookup` делают то же самое, но вместо

того, чтобы создавать простой список элементов, оба выдают результаты, которые поддерживают ассоциативный поиск. `ToDictionary` возвращает `IDictionary< TKey, TValue >`, поэтому он предназначен для сценариев, где ключ соответствует ровно одному значению. `ToLookup` разработан для сценариев, в которых ключ может быть связан с несколькими значениями, поэтому возвращает другой тип, а именно `ILookup< TKey, TValue >`.

Я не упоминал этот интерфейс в главе 5, потому что он специфичен для LINQ. По сути, он очень похож на интерфейс словаря, за исключением того, что индексатор возвращает `IEnumerable< TValue >` вместо единственного `TValue`.

Хотя преобразования массива и списка не принимают аргументов, преобразованиям словаря и поиска необходимо указать, какое значение использовать в качестве ключа для каждого исходного элемента. Для этого вы передаете им лямбду, как показано в листинге 10.65. В качестве ключа он использует свойство курса `Category`.

Листинг 10.65. Создание поиска

```
ILookup<string, Course> categoryLookup =
    Course.Catalog.ToLookup(course => course.Category);
foreach (Course c in categoryLookup["MAT"])
{
    Console.WriteLine(c.Title);
}
```

Оператор `ToDictionary` имеет перегрузку, которая принимает тот же аргумент, но возвращает словарь вместо поиска. Если вы вызовете его так же, как я вызывал `ToLookup` в листинге 10.65, будет вызвано исключение, потому что несколько объектов курса имеют общие категории, поэтому они будут соответствовать одному и тому же ключу. `ToDictionary` требует, чтобы каждый объект имел уникальный ключ. Чтобы создать словарь из каталога курсов, вам нужно либо сначала сгруппировать данные по категориям и каждая запись словаря должна относиться ко всей группе, либо использовать лямбда-выражение, возвращающее составной ключ, основанный на категории курса и номере, потому что эта комбинация уникальна для каждого курса.

Оба оператора также имеют перегрузки, которые принимают пару лямбд — одну, которая извлекает ключ, и вторую, которая выбирает, что использовать в качестве соответствующего значения (вы не обязаны использовать в качестве значения исходный элемент). Наконец, есть перегрузки, которые принимают `IEqualityComparer< T >`.

Теперь вы видели все стандартные операторы LINQ, но поскольку это заняло довольно много страниц, полезно иметь краткое резюме. Таблица 10.1 перечисляет операторы и кратко описывает, для чего они предназначены.

Таблица 10.1. Сводка операторов LINQ

Оператор	Назначение
Aggregate	С помощью предоставленной пользователем функции объединяет все элементы для выдачи одного результата
All	Возвращает <code>true</code> , если предоставленный предикат не является <code>false</code> ни для каких элементов
Any	Возвращает <code>true</code> , если предоставленный предикат имеет значение <code>true</code> хотя бы для одного элемента
Append	Возвращает последовательность из всех элементов входной последовательности с добавлением одного элемента в ее конец
AsEnumerable	Возвращает последовательность в виде <code>IEnumerable<T></code> . (Пригодится для принудительного использования LINQ to Objects.)
AsParallel	Возвращает <code>ParallelQuery<T></code> для параллельного выполнения запроса
AsQueryable	Где это доступно, обеспечивает использование обработки <code>IQueryable<T></code>
Average	Вычисляет среднее арифметическое элементов
Cast	Приводит каждый элемент последовательности к указанному типу
Concat	Формирует последовательность путем объединения двух последовательностей
Contains	Возвращает <code>true</code> , если указанный элемент содержится в последовательности
Count, LongCount	Возвращает количество элементов в последовательности
DefaultIfEmpty	Выдает элементы исходной последовательности, а если их нет, то создает единственный элемент со значением по умолчанию для типа элемента
Distinct	Удаляет повторяющиеся значения
ElementAt	Возвращает элемент в указанной позиции (выдает исключение, если выходит за пределы диапазона)
ElementAtOrDefault	Возвращает элемент в указанной позиции (производит значение по умолчанию для типа элемента, если он находится вне диапазона)
Except	Отфильтровывает элементы, содержащиеся в другой предоставленной коллекции
First	Возвращает первый элемент, выдавая исключение, если элементов нет

Оператор	Назначение
FirstOrDefault	Возвращает первый элемент или значение по умолчанию для типа элемента, если элементы отсутствуют
GroupBy	Собирает элементы в группы
GroupJoin	Группирует элементы в другой последовательности, согласно тому как они связаны с элементами во входной последовательности
Intersect	Отфильтровывает элементы, которых нет в другой коллекции
Join	Создает элемент для каждой подходящей пары элементов из двух входных последовательностей
Last	Возвращает последний элемент, вызывая исключение, если элементы отсутствуют
LastOrDefault	Возвращает последний элемент или значение по умолчанию для типа элемента, если элементы отсутствуют
Max	Возвращает наибольшее значение
Min	Возвращает наименьшее значение
OfType	Отфильтровывает элементы, которые не относятся к указанному типу
OrderBy	Выдает элементы в порядке возрастания
OrderByDescending	Выдает элементы в порядке убывания
Prepend	Возвращает последовательность, начинающуюся с указанного отдельного элемента, за которым следуют все элементы из входной последовательности
Reverse	Выводит элементы в порядке, обратном вводу
Select	Проектирует каждый элемент через функцию
SelectMany	Объединяет несколько коллекций в одну
SequenceEqual	Возвращает <code>true</code> , только если все элементы равны элементам в другой предоставленной последовательности
Single	Возвращает единственный элемент и выдает исключение, если в результате не найдено ни одного элемента или найдено более одного
SingleOrDefault	Возвращает единственный элемент или значение по умолчанию для типа элемента, если элементы отсутствуют. Выдает исключение, если найдено более одного элемента
Skip	Отфильтровывает указанное количество элементов от начала
SkipWhile	Отфильтровывает элементы от начала до тех пор, пока они соответствуют предикату
Sum	Возвращает результат сложения всех элементов

Оператор	Назначение
Take	Выдаёт указанное количество элементов, отбрасывая остальные
TakeLast	Выдаёт указанное количество элементов от конца ввода (отбрасывая все предыдущие)
TakeWhile	Выдаёт элементы до тех пор, пока они соответствуют предикату, отбрасывая остальную часть последовательности, как только один из них не соответствует
ToArray	Возвращает массив, содержащий все элементы
ToDictionary	Возвращает словарь, содержащий все элементы
ToHashSet	Возвращает HashSet<T>, содержащий все элементы
ToList	Возвращает List<T>, содержащий все элементы
ToLookup	Возвращает многозначный ассоциативный поиск, содержащий все элементы
Union	Выдаёт все элементы, которые содержатся в одной или обеих входных последовательностях
Where	Отфильтровывает элементы, которые не соответствуют предоставленному предикату
Zip	Объединяет пары предметов из двух входов

Генерирование последовательностей

Класс `Enumerable` определяет методы расширения для `IEnumerable<T>`, которые составляют LINQ to Objects. Он также содержит несколько дополнительных статических методов (не методов расширения), которые можно использовать для создания новых последовательностей. `Enumerable.Range` принимает два аргумента `int` и возвращает `IEnumerable<int>`, выдающий постоянно увеличивающуюся последовательность чисел, начинающуюся со значения первого аргумента и содержащую количество чисел, равное второму аргументу. Например, `Enumerable.Range (15, 10)` создает последовательность, содержащую числа от 15 до 24 (включительно).

`Enumerable.Repeat<T>` принимает значение типа `T` и количество. При этом возвращается последовательность, которая будет выдавать это значение указанное количество раз.

`Enumerable.Empty<T>` возвращает `IEnumerable<T>`, которая это не содержит элементов. Это может показаться не очень полезной функцией, потому что есть гораздо менее многословная альтернатива. Вы можете написать

оператор `new T[0]`, который создает массив без элементов. (Массивы типа `T` реализуют `IEnumerable<T>`.) Однако преимущество `Enumerable.Empty<T>` в том, что для любого `T` он каждый раз возвращает один и тот же экземпляр. Это означает, что, если по какой-либо причине вам понадобится пустая последовательность в цикле, который выполняет много итераций, `Enumerable.Empty<T>` будет более эффективным решением, потому что это снизит нагрузку на сборщик мусора.

Другие реализации LINQ

В большинстве примеров этой главы я использовал LINQ to Objects, за исключением нескольких, которые ссылались на Entity Framework. В этом разделе я приведу краткое описание ряда других технологий на основе LINQ. Это далеко не полный список, поскольку провайдер LINQ может написать любой разработчик.

Entity Framework

Примеры работы с базой данных, которые я показал, использовали провайдер LINQ, являющийся частью Entity Framework (EF). EF – это технология доступа к данным, которая поставлялась как часть .NET Framework, но теперь перенесена в отдельный пакет NuGet, `Microsoft.EntityFrameworkCore`. (Старые версии все еще встроены в .NET Framework, но не в .NET Core. В любом случае, если вы хотите использовать последнюю версию, вы должны использовать пакет NuGet.) EF может устанавливать соответствие между базой данных и уровнем объекта. Она поддерживает несколько провайдеров баз данных и работает на основе `IQueryable<T>`.

Для каждого постоянного типа объекта в модели данных EF способна предоставить объект, который реализует `IQueryable<T>`, что можно использовать в качестве отправной точки для построения запросов по извлечению сущностей этого типа и связанных типов. Поскольку `IQueryable<T>` не является уникальным для EF, вы будете использовать стандартный набор методов расширения, предоставляемых классом `Queryable` в пространстве имен `System.Linq`, но этот механизм предназначен для того, чтобы каждый провайдер мог подключать собственное поведение.

Из-за того что `IQueryable<T>` определяет операторы LINQ как методы, которые принимают аргументы `Expression<T>` вместо простых типов делегатов,

любые выражения, которые вы записываете в выражениях запроса или в качестве лямбда-аргументов для базовых методов операторов, превратятся в генерированный компилятором код, который создает дерево объектов, представляющее структуру выражения. EF использует, чтобы иметь возможность генерировать запросы к базе данных, которые получают нужные вам данные. Это же означает, что вы обязаны использовать лямбды. В отличие от LINQ to Objects, в запросе EF вы не можете использовать анонимные методы или делегаты.



Из-за того что `IQueryable<T>` наследуется от `IEnumerable<T>`, можно использовать операторы `LINQ to Objects` для любого источника EF. Вы можете делать это явно с использованием оператора `AsEnumerable<T>`, но это также может произойти случайно, если вы использовали перегрузку, которая поддерживается `LINQ to Objects`, но не `IQueryable<T>`. Например, если вы попытаетесь использовать делегат вместо лямбды в качестве, скажем, предиката для оператора `Where`, это приведет к откату к `LINQ to Objects`. В результате EF загрузит все содержимое таблицы, а затем выполнит оператор `Where` на стороне клиента. Это вряд ли можно считать хорошим исходом.

Parallel LINQ (PLINQ)

Parallel LINQ похож на LINQ to Objects тем, что основан на объектах и делегатах, а не на деревьях выражений и трансляции запросов. Но когда вы захотите получить результаты запроса, он будет по возможности запускать многопоточное выполнение, пытаясь с помощью пула потоков использовать доступные ресурсы ЦП более эффективно. В главе 16 посмотрим на многопоточность в действии.

LINQ to XML

LINQ to XML не является провайдером LINQ. Я упоминаю о нем лишь потому, что его название звучит как название провайдера. В действительности это API для создания и анализа XML-документов. Он называется LINQ to XML, потому что был разработан, чтобы упростить выполнение запросов LINQ к документам XML, но это достигается путем представления документов XML через объектную модель .NET. Библиотека классов .NET предоставляет для этого два отдельных API: помимо LINQ to XML она также предлагает

XML Document Object Model (DOM). DOM основана на платформонезависимом стандарте и как результат не вполне соответствует идиомам .NET и выглядит излишне причудливо по сравнению с большинством библиотек классов. LINQ to XML разработан исключительно для .NET, поэтому лучше работает с обычными методами C#. Это включает в себя хорошую работу с LINQ, что обеспечивается методами, извлекающими функции из документа в форме `IEnumerable<T>`. Это позволяет полагаться на LINQ to Objects для определения и выполнения запросов.

Реактивные расширения

Реактивные расширения для .NET (или Rx, как их часто сокращают) являются предметом следующей главы, поэтому я не буду углубляться в них здесь, но они являются хорошей иллюстрацией того, как операторы LINQ могут работать с большим разнообразием типов. Rx инвертирует модель, показанную в этой главе, в который мы запрашиваем элементы по мере собственной готовности. Таким образом, вместо написания цикла `foreach`, который выполняет итерацию по запросу, или вместо вызова одного из операторов, выполняющих запрос, вроде `ToArray` или `SingleOrDefault`, мы ждем, пока источник Rx обратится к нам, когда будет готов предоставить данные.

Несмотря на эту инверсию, есть провайдер LINQ для Rx, который поддерживает большинство стандартных операторов LINQ.

Tx (LINQ to Logs and Traces)

Одним из наименее известных провайдеров LINQ является Tx, который поддерживает выполнение запросов LINQ непосредственно в файлах журналов и файлах Event Tracing for Windows (ETW). Это проект с открытым исходным кодом, написанный Microsoft и доступный по адресу <https://github.com/Microsoft/Tx>.

Итог

В этой главе я показал синтаксис запроса, который поддерживает ряд наиболее часто используемых функций LINQ. Это позволяет писать на C# запросы, которые напоминают запросы к базе данных, но могут обращаться

к любому провайдеру LINQ, включая LINQ to Objects, что позволяет нам выполнять запросы к собственным объектным моделям. Я показал стандартные операторы LINQ для запросов, все из которых доступны для LINQ to Objects и большинство из которых доступны для провайдеров баз данных. Я также дал краткий обзор некоторых распространенных провайдеров LINQ для приложений .NET.

Последним упомянутым провайдером был Rx. Но прежде, чем рассмотреть провайдер LINQ Rx, следующую главу посвятим тому, как работает сам Rx.

ГЛАВА 11

Реактивные расширения

Реактивные расширения (Reactive Extensions) для .NET или Rx предназначены для работы с асинхронными и основанными на событиях источниками информации. Rx предоставляет сервисы, которые помогут вам организовать и синхронизировать реакцию вашего кода на данные из таких источников. В главе 9 мы уже видели, как определять и подписываться на события, но Rx предлагает гораздо больше этих базовых функций. Он предлагает абстракцию для источников событий, освоить работу с которой сложнее, чем работу с событиями. Но она поставляется с мощным набором операторов, который делает объединение и управление несколькими потоками событий гораздо проще, чем это было бы возможно с помощью публичных методов, предоставляемых делегатами и событиями .NET.

Фундаментальная абстракция Rx, `IObservable<T>`, представляет последовательность элементов, а ее операторы определены как методы расширения для этого интерфейса. Это выглядит очень похоже на LINQ to Objects, и сходства действительно есть — не только в том, что `IObservable<T>` имеет много общего с `IEnumerable<T>`, но и в том, что Rx поддерживает почти все стандартные операторы LINQ. Если вы знакомы с LINQ to Objects, вы будете чувствовать себя как рыба в воде и с Rx. Разница в том, что в Rx последовательности менее пассивны. В отличие от `IEnumerable<T>`, источники Rx не ждут, когда у них запросят элементы, а пользователь источника Rx не может потребовать предоставления следующего элемента. Вместо этого Rx использует push-модель, в которой источник уведомляет своих получателей о доступности элементов.

Например, если вы пишете приложение, которое имеет дело с финансовой информацией в реальном времени, такой как данные о рыночных ценах, `IObservable<T>` — это гораздо более подходящая модель, чем `IEnumerable<T>`. Поскольку Rx реализует стандартные операторы LINQ, вы можете писать запросы к действующему источнику — вы можете сузить поток событий с помощью выражения `where` или сгруппировать их по символу акций. Rx выходит за рамки стандартного LINQ, добавляя собственные операторы,

которые учитывают временную природу действующего источника событий. Например, вы можете написать запрос, который предоставляет данные только для акций, цены на которые меняются чаще, чем минимальная ставка.

Push-ориентированный подход Rx делает его более подходящим для событийных источников, нежели `IEnumerable<T>`. Но почему бы просто не использовать события или даже простые делегаты? Rx устраняет четыре недостатка этих альтернатив. Во-первых, он определяет стандартный способ для источников сообщать об ошибках. Во-вторых, он способен поставлять элементы в четко определенном порядке, даже в многопоточных сценариях с участием многочисленных источников. В-третьих, Rx дает четкий способ сигнализировать, когда больше нет элементов. В-четвертых, поскольку традиционное событие представлено элементом особого типа, а не обычным объектом, существуют значительные ограничения в отношении того, что вы можете проделать с событием. Например, вы не можете передать событие в качестве аргумента методу. Rx превращает источник события в полноправную сущность, потому что это просто объект. Это означает, что вы можете передать источник события в качестве аргумента, сохранить его в поле или передать в свойстве — все, что нельзя сделать с обычным событием .NET. Конечно, вы можете передать в качестве аргумента делегат, но это не одно и то же: делегаты обрабатывают события, но не представляют их. Нет способа написать метод, который подписывается на какое-то событие .NET, которое вы передаете в качестве аргумента, потому что нельзя передавать само фактическое событие. Rx исправляет это, представляя источники событий как объекты, а не особую самобытную особенность системы типов, которая работает не так, как все остальное.

Речь, конечно, обо всех тех функциях, которые вы легко можете задействовать с `IEnumerable<T>`. Коллекция может просто генерировать исключение, когда перебирается ее содержимое, но с обратными вызовами гораздо менее очевидно, когда и где нужно вызывать исключения. `IEnumerable<T>` заставляет извлекать элементы по одному, поэтому порядок однозначен, но в случае простых событий и делегатов это не обязательно. Кроме того, `IEnumerable<T>` сообщает, если был достигнут конец коллекции, но с простым обратным вызовом не всегда ясно, когда произошел последний вызов. `IObservable<T>` предоставляет все эти возможности, принося в мир событий то, что мы принимаем как должное в случае `IEnumerable<T>`.

Представляя согласованную абстракцию, которая решает все эти проблемы, Rx использует все преимущества LINQ для основанных на событиях

сценариев. Rx не заменяет события, иначе я бы не посвятил им пятую часть главы 9. На самом деле Rx способна интегрироваться с событиями. Rx может связывать свои собственные абстракции с рядом других, не только с обычными событиями, но и с `IEnumerable<T>` и с различными моделями асинхронного программирования. Не делая события устаревшими, Rx поднимает их возможности на новый уровень. В Rx разобраться намного сложнее, чем в событиях, но если вы это сделаете, то у вас на руках будет мощный инструмент.

Два интерфейса — основа Rx. Источники, которые представляют элементы посредством этой модели, реализуют `IObservable<T>`. Обязанность подписчиков — предоставить объект, реализующий `IObserver<T>`. Оба этих интерфейса встроены в .NET. Другие части Rx находятся в пакете NuGet `System.Reactive`.

Ключевые интерфейсы

Два наиболее важных типа в Rx — это интерфейсы `IObservable<T>` и `IObserver<T>`. Они настолько важны, что находятся в пространстве имен `System`. Листинг 11.1 показывает их определения.

Листинг 11.1. `IObservable<T>` и `IObserver<T>`

```
public interface IObserver<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<in T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}
```

Ключевая абстракция Rx, `IObservable<T>`, реализуется источниками событий. Вместо использования ключевого слова `event` источник выстраивает события как последовательность элементов. `IObservable<T>` предоставляет элементы подписчикам по мере готовности.

Как видите, аргумент типа для `IObservable<T>` является ковариантным, что означает, что если у вас есть тип `Base`, который является базовым типом дру-

гого типа `Derived`, то точно так же, как вы можете передать `Derived` в любой метод, ожидающий `Base`, вы можете передать `IObservable<Derived>` туда, где ожидается `IObservable<Base>`. Интуитивно здесь ожидаешь увидеть ключевое слово `out`, потому что, как и в случае `IEnumerable<T>`, это источник информации — из него выходят элементы. И наоборот, элементы отправляются в реализацию `IObserver<T>` подписчика, так что здесь имеется ключевое слово `in`, которое обозначает контравариантность — вы можете передать `IObserver<Base>` чему-либо ожидающему `IObserver<Derived>`. (Вариантность описана в главе 6.)

Мы можем подписаться на источник, передав реализацию `IObserver<T>` в метод `Subscribe`. Источник будет вызывать `OnNext`, когда нужно сообщить о событиях, и он может вызвать `OnCompleted`, чтобы сообщить, что больше не будет никакой активности. Если источник хочет сообщить об ошибке, он может вызвать `OnError`. И `OnCompleted`, и `OnError` обозначают конец потока — после них наблюдаемый не должен вызывать какие-либо дополнительные методы наблюдателя.



Нарушив эти правила, вы не обязательно немедленно получите исключение. В некоторых случаях так и будет — если вы используете библиотеку NuGet `System.Reactive`, чтобы реализовать и использовать эти интерфейсы, то существуют определенные обстоятельства, в которых она может обнаружить такую ошибку. Но в целом именно код должен следить за тем, чтобы вызовы этих методов соответствовали правилам.

Существует соглашение о визуальном представлении работы Rx. Его иногда называют шариковой диаграммой, потому что она состоит в основном из маленьких кружков, которые напоминают стеклянные шарики. В рис. 11.1 это соглашение используется для представления двух последовательностей событий. Горизонтальные линии представляют подписки на источники, вертикальная полоса слева указывает на начало подписки, а горизонтальная позиция — на то, что что-то произошло (затраченное время увеличивается слева направо). Кружки указывают на вызовы `OnNext`, т. е. события, о которых сообщает источник). Стрелка на правом конце говорит о том, что подписка все еще активна к концу отрезка времени, представленного на диаграмме. Вертикальная полоса справа указывает на конец подписки — либо из-за вызова `OnError` или `OnCompleted`, либо из-за того, что подписчик отписался.

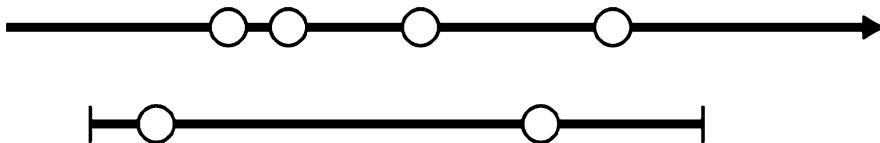


Рис. 11.1. Простая Marble Diagram («шариковая диаграмма»)

Когда вы вызываете `Subscribe` для наблюдаемого объекта, он возвращает объект, который реализует `IDisposable`, что дает способ отписаться. Если вы вызываете `Dispose`, наблюдаемый объект больше не будет отправлять уведомления вашему наблюдателю. Это может быть более удобным, чем механизм отказа от подписки на событие; чтобы отказаться от подписки на событие, вы должны передать делегат, эквивалентный тому, который вы использовали для подписки. Если вы используете анонимные методы, выглядеть это может на удивление неуклюже, так как зачастую единственный способ сделать это — сохранить ссылку на исходный делегат. При использовании Rx любая подписка на источник представляется в виде `IDisposable`, что упрощает единообразную обработку. Вообще, не так часто нужно от чего-либо отписываться — это необходимо только в том случае, если вы хотите прекратить получать уведомления до завершения работы источника (что является примером относительно необычной в .NET выборочной удалаемости).

IObserver<T>

Как вы увидите, на практике мы нечасто напрямую вызываем метод источника `Subscribe`, как и самостоятельно реализуем `IObserver<T>`. Вместо этого обычно используют один из содержащихся в Rx и основанных на делегатах методов расширения, который и присоединяет предоставляемую Rx реализацию. Однако эти методы расширения не являются частью фундаментальных типов Rx, поэтому сейчас я покажу, что нужно написать, если эти интерфейсы — это все, что у вас есть. Листинг 11.2 показывает простой, но полноценный класс-наблюдатель.

Источники Rx (т. е. реализации `IObservable<T>`) должны предоставить определенные гарантии относительно того, как вызывают методы наблюдателя. Как я уже упоминал, вызовы происходят в определенном порядке: `OnNext` вызывается для каждого предоставляемого источником элемента, но, как только вызывается `OnCompleted` или `OnError`, наблюдатель понимает, что

больше не будет вызовов ни одного из трех методов. Любой из этих методов сигнализирует об окончании последовательности.

Листинг 11.2. Простая реализация `IObserver<T>`

```
class MySubscriber<T> : IObserver<T>
{
    public void OnNext(T value) => Console.WriteLine("Received: " +
value);
    public void OnCompleted() => Console.WriteLine("Complete");
    public void OnError(Exception ex) => Console.WriteLine("Error: " + ex);
}
```

Кроме того, вызовам не разрешается перекрываться — когда наблюдаемый источник вызывает один из методов нашего наблюдателя, он должен дождаться возврата этого метода, прежде чем снова его вызывать. Многопоточный наблюдаемый объект должен позаботиться о координации своих вызовов, и даже в однопоточном мире возможность рекурсии может сделать необходимостью обнаружение и предотвращение повторных входящих вызовов в источнике.

Это упрощает жизнь наблюдателя. Поскольку Rx предоставляет события в виде последовательности, моему коду не нужно разбираться с одновременными вызовами. Вызов методов в правильном порядке — обязанность источника. Итак, хотя `Iobservable<T>` может выглядеть как более простой интерфейс, имеющий только один метод, он более требовательный к реализации. Как вы увидите позже, обычно проще позволить библиотекам Rx реализовать его за вас. Однако все же важно знать, как работают наблюдаемые источники, поэтому я начну с реализации его вручную.

Iobservable<T>

Rx различает горячие (активные) и холодные (пассивные) наблюдаемые источники. Горячий выдаст каждое значение по мере того, как происходит что-то интересное, и если в этот момент нет подключенных подписчиков, это значение будет потеряно. Горячий наблюдаемый источник, как правило, представляет что-то действующее, например ввод с помощью мыши, нажатие клавиш или данные, сообщаемые датчиком. Поэтому значения, которые он выдает, не зависят от того, сколько подписчиков подключено (и есть ли они вообще). Горячие источники обычно ведут себя по типу трансляции — отправляют каждый элемент всем своим подписчикам. Такой источник может

быть более сложен для реализации, поэтому сначала я расскажу о холодных источниках.

Реализация холодных источников

В то время как горячие источники сообщают о чем-то тогда, когда пожелают, холодные наблюдаемые источники действуют иначе. Они начинают выдавать значения, когда подписчик подписывается, и отправляют значения каждому подписчику отдельно, а не в рамках широковещательной рассылки. Это означает, что подписчик ничего не пропустит, если опаздывает, так как источник начинает выдавать элементы, когда вы подписываетесь. Листинг 11.3 показывает очень простой холодный источник.

Листинг 11.3. Простой холодный наблюдаемый источник

```
public class SimpleColdSource : IObservable<string>
{
    public IDisposable Subscribe(IObserver<string> observer)
    {
        observer.OnNext("Hello,");
        observer.OnNext("world!");
        observer.OnCompleted();
        return NullDisposable.Instance;
    }

    private class NullDisposable : IDisposable
    {
        public readonly static NullDisposable Instance =
            new NullDisposable();
        public void Dispose() { }
    }
}
```

В тот момент, когда наблюдатель подписывается, источник предоставляет два значения, строки "Hello" и "world!", после чего указывает на конец последовательности, вызывая `OnCompleted`. Он делает все это внутри `Subscribe`, так что на самом деле это не похоже на подписку — последовательность уже завершена к моменту завершения `Subscribe`, так что для отмены подписки не нужно проделывать никаких осмысленных шагов. Вот почему пример возвращает пустую реализацию `IDisposable`. (Я выбрал очень простой пример, чтобы показать базу. Реальные источники будут гораздо сложнее.)

Чтобы показать все это в действии, нам придется создать экземпляр `SimpleColdSource`, а также экземпляр моего класса-наблюдателя из листинга 11.2 и использовать его для подписки на источник, как это делает листинг 11.4.

Листинг 11.4. Прикрепление наблюдателя к наблюдаемому источнику

```
var source = new SimpleColdSource();
var sub = new MySubscriber<string>();
source.Subscribe(sub);
```

Вполне предсказуемо, это дает следующий результат:

```
Received: Hello,
Received: world!
Complete
```

В общем, холодный наблюдатель будет иметь доступ к некоторому базовому источнику информации, которую он может выдавать подписчику по требованию. В листинге 11.3 этот «источник» был просто двумя жестко закодированными значениями. Листинг 11.5 показывает немного более интересный наблюдаемый холодный источник, который считывает строки из файла и предоставляет их подписчику.

Листинг 11.5. Холодный наблюдаемый источник, представляющий содержимое файла

```
public class FilePusher : IObservable<string>
{
    private readonly string _path;
    public FilePusher(string path)
    {
        _path = path;
    }

    public IDisposable Subscribe(IObserver<string> observer)
    {
        using (var sr = new StreamReader(_path))
        {
            while (!sr.EndOfStream)
            {
                observer.OnNext(sr.ReadLine());
            }
        }
        observer.OnCompleted();
        return NullDisposable.Instance;
    }

    private class NullDisposable : IDisposable
    {
        public static NullDisposable Instance = new NullDisposable();
        public void Dispose() { }
    }
}
```

Как и раньше, он не является действующим источником событий и начинает работать только тогда, когда что-то на него подписывается, но все же он немного интереснее, чем источник в листинге 11.3. Он обращается к наблюдателю по мере извлечения каждой строки из файла, поэтому, хотя точка, с которой он начинает свою работу, определяется подписчиком, именно источник контролирует скорость, с которой предоставляются значения. Как и в листинге 11.3, он доставляет все элементы наблюдателю в потоке вызывающей стороны в вызове `Subscribe`. Но концептуальный скачок от листинга 11.5 к тому, в котором код чтения из файла либо выполнялся в отдельном потоке, либо использовал асинхронные методы (например, описанные в главе 17), относительно небольшой. Это позволяло `Subscribe` возвращаться до завершения работы (после чего вам следовало написать более интересную реализацию `IDisposable`, чтобы вызывающая сторона имела возможность отписаться). Это все еще холодный источник, потому что он представляет некоторый базовый набор данных, который он может перечислять с самого начала в интересах каждого отдельного подписчика.

Листинг 11.5 не совсем завершен — он не обрабатывает ошибки, возникающие при чтении из файла. Их нужно отслеживать, после чего вызывать метод `OnError` наблюдателя. К сожалению, этого не добиться заключением всего цикла в блок `try`, поскольку он также будет отлавливать исключения, возникающие в методе `OnNext` наблюдателя. Если он вызывает исключение, мы должны позволить ему подняться дальше по стеку — нам следует обрабатывать только те исключения, которые возникают в ожидаемых местах нашего кода. К сожалению, это серьезно усложняет код. Листинг 11.6 помещает весь код, который использует `FileStream`, в блок `try`, но позволяет любым исключениям, генерируемым наблюдателем, двигаться вверх по стеку, потому что их обработка — это не наше дело.

Листинг 11.6. Обработка ошибок файловой системы, но не ошибок наблюдателя

```
public IDisposable Subscribe(IObserver<string> observer)
{
    StreamReader sr = null;
    string line = null;
    bool failed = false;

    try
    {
        while (true)
        {
```

```
try
{
    if (sr == null)
    {
        sr = new StreamReader(_path);
    }
    if (sr.EndOfStream)
    {
        break;
    }
    line = sr.ReadLine();
}
catch (IOException x)
{
    observer.OnError(x);
    failed = true;
    break;
}

observer.OnNext(line);
}
}
finally
{
    if (sr != null)
    {
        sr.Dispose();
    }
}
if (!failed)
{
    observer.OnCompleted();
}
return NullDisposable.Instance;
}
```

Если при чтении из файла возникают исключения ввода-вывода, сообщение о них передается посредством метода `OnError` наблюдателя. Таким образом, этот источник использует все три метода `IObserver<T>`.

Реализация горячих источников

Горячие источники уведомляют всех имеющихся подписчиков о значениях, как только они становятся доступными. Это означает, что любой горячий наблюдаемый источник должен знать, какие наблюдатели в настоящее время

на него подписаны. Горячие источники разделяют подписку и уведомление способом, который холодные источники обычно не используют.

Листинг 11.7 – это наблюдаемый источник, который передает один элемент для каждого нажатия клавиши, и для горячего источника он крайне простой. Он однопоточный, поэтому ему не требуется делать ничего особенного для избежания перекрывающихся вызовов. Он не сообщает об ошибках, поэтому ему никогда не требуется вызывать методы `OnError` наблюдателей. И он никогда не останавливается, поэтому ему также не нужно вызывать `OnCompleted`. Несмотря на это, он довольно сложен. (Когда я перейду к поддержке библиотеки Rx, все станет намного проще — этот пример довольно сложный, потому что пока я использую только два ключевых интерфейса.)

Листинг 11.7. `IObservable<T>` для отслеживания нажатий клавиш

```
public class KeyWatcher : IObservable<char>
{
    private readonly List<Subscription> _subscriptions =
        new List<Subscription>();

    public IDisposable Subscribe(IObserver<char> observer)
    {
        var sub = new Subscription(this, observer);
        _subscriptions.Add(sub);
        return sub;
    }

    public void Run()
    {
        while (true)
        {
            // Передача сюда true остановит вывод символа консолью
            char c = Console.ReadKey(true).KeyChar;
            // Итерация по снимку состояния, чтобы обработать случай,
            // когда наблюдатель подписывается изнутри своего метода
            // OnNext.
            foreach (Subscription sub in _subscriptions.ToArray())
            {
                sub.Observer.OnNext(c);
            }
        }
    }

    private void RemoveSubscription(Subscription sub)
    {
```

```
        _subscriptions.Remove(sub);
    }

private class Subscription : IDisposable
{
    private KeyWatcher _parent;
    public Subscription(KeyWatcher parent, IObserver<char> observer)
    {
        _parent = parent;
        Observer = observer;
    }

    public IObserver<char> Observer { get; }

    public void Dispose()
    {
        if (_parent != null)
        {
            _parent.RemoveSubscription(this);
            _parent = null;
        }
    }
}
```

Листинг определяет вложенный класс с именем `Subscription` для отслеживания каждого наблюдателя, который подписывается. Кроме того, он содержит реализацию `IDisposable`, которую должен возвращать наш метод `Subscribe`. Наблюдаемый объект создает новый экземпляр этого вложенного класса и добавляет его в список текущих подписчиков во время выполнения `Subscribe`, а затем, если вызывается `Dispose`, удаляет себя из этого списка.

Как правило, в .NET после окончания использования следует вызывать `Dispose` любых `IDisposable`, выделенных от вашего имени. Однако в Rx обычно не удаляют объекты, представляющие подписки, поэтому, если вы реализуете такой объект, вы не должны рассчитывать на его удаление. Обычно это не нужно, потому что Rx способен выполнить очистку за вас. В отличие от обычных событий .NET или делегатов, наблюдаемые объекты могут однозначно завершать работу, и в этот момент любые ресурсы, выделенные подписчикам, могут быть освобождены. (Некоторые работают неопределенно долго, но в этом случае подписки обычно остаются активными в течение всей жизни программы.) Следует признать, что приведенные мной примеры не освобождаются автоматически, потому что мои реализации до-

статочно простые, чтобы в этом не нуждаться, но библиотеки Rx это делают, если вы используете их реализации источников и подписчиков. Единственный случай, когда вы избавляетесь от подписки в Rx, — это если вы хотите отменить подписку до завершения работы источника.



Подписчики не обязаны следить за тем, чтобы `object`, возвращаемый `Subscribe`, оставался достижимым. Если вам не нужна возможность отписываться раньше времени, вы можете просто проигнорировать его, и не будет иметь значения, освободил ли сборщик мусора этот объект. Ни одна из реализаций `IDisposable`, которые Rx предоставляет для подписок, не имеет финализатора. (И хотя их обычно не нужно реализовывать самостоятельно — я делаю это здесь только для иллюстрации того, как это работает, — если вы все же решили написать свою собственную, используйте тот же подход: не реализуйте финализатор в классе, который представляет подписку.)

Класс `KeyWatcher` в листинге 11.7 содержит метод `Run`. Это не стандартная функция Rx, а обычный цикл, который ожидает ввода с клавиатуры, — данный наблюдаемый объект не будет генерировать никаких уведомлений, пока что-то не вызовет этот метод. Каждый раз, когда этот цикл получает клавишу, он вызывает метод `OnNext` для каждого подписанного в настоящее время наблюдателя. Обратите внимание, что я создаю копию списка подписчиков (вызов `ToArrayList` — это простой способ получить `List<T>`, дублирующий его содержимое), потому что есть все шансы, что подписчик может отказаться от подписки в середине вызова `OnNext`, а это означает, что если я передам список подписчиков напрямую в `foreach`, я получу исключение. Это связано с тем, что списки не позволяют добавлять и удалять элементы, если вы выполняете их итерацию.



Этот пример защищен только от повторных вызовов в том же потоке; обработка многопоточной отписки будет в целом более сложной. Но даже создания копии недостаточно в полной мере. В действительности следует проверять, что каждый наблюдатель в моем снимке данных все еще подписан, прежде чем вызывать его `OnNext`, потому что вполне возможно, что один наблюдатель может отписать другого наблюдателя. Пример также не предусматривает возможности отписки из другого потока. Позже я заменю все это гораздо более надежной реализацией из библиотеки Rx.

В использовании этот горячий источник очень похож на мои холодные источники. Нам нужно создать экземпляр `KeyWatcher`, а также другой экземпляр моего класса наблюдателя (на этот раз с аргументом типа `char`, поскольку этот источник генерирует символы вместо строк). Поскольку этот источник не генерирует элементы до тех пор, пока не будет запущен его цикл отслеживания, для его запуска мне необходимо вызвать `Run`, как это сделано в листинге 11.8.

Листинг 11.8. Прикрепление наблюдателя к наблюдаемому источнику

```
var source = new KeyWatcher();
var sub = new MySubscriber<char>();
source.Subscribe(sub);
source.Run();
```

После запуска этого кода приложение будет ожидать ввода с клавиатуры, и, если вы нажмете, скажем, клавишу *m*, наблюдатель (листинг 11.2) отобразит сообщение `Received: m.` (И поскольку мой источник работает бесконечно, метод `Run` никогда не завершится.)

Возможно, вам придется иметь дело со смесью горячих и холодных наблюдаемых источников. Кроме того, некоторые холодные источники имеют какие-то характеристики горячих. Например, можно вообразить источник, представляющий предупреждения, и, возможно, имеет смысл реализовать его так, чтобы он сохранял предупреждения. Это поможет убедиться, что вы не пропустили ничего, что произошло между созданием источника и подключением подписчика. Таким образом, это был бы холодный источник — любой новый подписчик получал бы все события, произошедшие до подписки. Но после появления подписчика текущее поведение становилось бы больше похожим на горячий источник, потому что любые новые события транслировались бы всем текущим подписчикам. Как вы увидите, библиотеки Rx предоставляют различные способы смешивания и настройки двух типов источников.

Хотя полезно заглянуть в то, что приходится делать наблюдателям и наблюдаемым, более продуктивным будет позволить Rx проделать всю тяжелую работу. Поэтому сейчас я покажу, как создавать источники и подписчиков, если вместо двух основных интерфейсов можно использовать библиотеку NuGet `System.Reactive`.

Публикация и подписка с использованием делегатов

Если вы используете пакет NuGet `System.Reactive`, не нужно напрямую реализовывать ни `IObservable<T>`, ни `IObserver<T>`. Сама библиотека предоставляет несколько реализаций. Некоторые из них являются адаптерами, соединяющими другие представления асинхронно генерируемых последовательностей. Некоторые служат обертками для существующих наблюдаемых потоков. Но эти вспомогательные средства предназначены не только для адаптации существующих сущностей. Они также могут пригодиться, если вам нужен код, который создает новые элементы или действует как конечный пункт назначения элементов. Простейшие из этих помощников предоставляют API на основе делегатов для создания и использования наблюдаемых потоков.

Создание наблюдаемого источника с помощью делегатов

Как вы видели в некоторых из предыдущих примеров, `IObservable<T>` — это простой интерфейс, но источникам, которые его реализуют, зачастую приходится проделывать немалую работу для отслеживания подписчиков. А ведь мы еще даже не взглянули на всю картину. Как вы увидите в разделе «Планировщики» на с. 641, источник часто вынужден принимать дополнительные меры для обеспечения хорошей интеграции с механизмами потоков Rx. К счастью, библиотеки Rx способны проделать часть этой работы за нас. В листинге 11.9 показано, как использовать статический метод `Create` класса `Observable` для реализации холодного источника. (Каждый вызов `GetFilePusher` будет создавать новый источник, так что это фактически фабричный метод.)

Пример похож на листинг 11.5, так как предоставляет наблюдаемый источник, который по очереди передает подписчикам каждую строку в файле. (Как и в листинге 11.5, для ясности я опустил обработку ошибок. На практике вам нужно было бы сообщать об ошибках так же, как это сделано в листинге 11.6.) Суть кода та же, но мне удалось обойтись только одним методом вместо целого класса, потому что теперь реализацию `IObservable<T>` предоставляет Rx. Каждый раз, когда наблюдатель подписывается на этот наблюдаемый объект, Rx осуществляет обратный вызов метода, который я передал в `Create`. Поэтому все, что мне нужно сделать, это написать код,

который предоставляет элементы. Кроме реализации внешнего класса, реализующего `IObservable<T>`, мне также удалось избавиться от вложенного класса, который реализует `IDisposable`. Метод `Create` позволяет нам вместо объекта возвращать делегат `Action`, и он будет вызывать его, если подписчик решит отказаться от подписки. Поскольку мой метод не завершается до тех пор, пока не закончит создание элементов, я не могу делать ничего полезного, поэтому просто возвращаю пустой метод.

Листинг 11.9. Наблюдаемый источник на основе делегатов

```
public static IObservable<string> GetFilePusher(string path)
{
    return Observable.Create<string>(observer =>
    {
        using (var sr = new StreamReader(path))
        {
            while (!sr.EndOfStream)
            {
                observer.OnNext(sr.ReadLine());
            }
        }
        observer.OnCompleted();
        return () => { };
    });
}
```

Поэтому я написал гораздо меньше кода, чем в листинге 11.5, но помимо упрощения моей реализации `Observable.Create` делает для нас две менее заметные вещи, которые не сразу очевидны из кода.

Во-первых, если подписчик отписался раньше, код корректно прекратит отправлять ему элементы, даже если я не добавил никакого специального кода для обработки этого. Когда наблюдатель подписывается на источник такого рода, Rx не передает `IObserver<T>` прямиком в наш обратный вызов. Аргумент `observer` во вложенном методе в листинге 11.9 относится к предоставленной Rx обертке. Если базовый наблюдатель отписывается, эта оболочка автоматически прекращает пересылку уведомлений. Мой цикл будет проходить через файл даже после того, как подписчик отписался, что расточительно. Тем не менее подписчик хотя бы не получит элементы после того, как попросит меня остановиться. (Вам может быть интересно, как подписчик вообще может отписаться, с учетом того, что мой код не возвращается, пока не закончит работу. Но в многопоточных сценариях еще

до завершения моего кода можно получить `IDisposable`, предоставляемый оберткой Rx, который и представляет подписку.)

Вы можете использовать Rx в сочетании с функциями асинхронного языка C# (в частности, ключевыми словами `async` и `await`) и реализовать версию листинга 11.9, которая не только более эффективно обрабатывает отписку, но и читает из файла асинхронно, т. е. подписка не должна быть блокирующей. Это значительно более эффективно, но код практически идентичен. Я не буду показывать функции асинхронного языка до главы 17, так что пока это может быть не до конца понятно, но если вам все-таки интересно, листинг 11.10 показывает, как это выглядит. Измененные строки выделены жирным шрифтом. (Опять же, это версия без обработки ошибок. Асинхронные методы могут обрабатывать исключения во многом так же, как и синхронные, поэтому вы можете разбираться с ошибками с помощью того же подхода, что использован в листинге 11.6.)

Листинг 11.10. Асинхронный источник

```
public static IObservable<string> GetFilePusher(string path)
{
    return Observable.Create<string>(async (observer, cancel) =>
    {
        using (var sr = new StreamReader(path))
        {
            while (!sr.EndOfStream && !cancel.IsCancellationRequested)
            {
                observer.OnNext(await sr.ReadLineAsync());
            }
        }
        observer.OnCompleted();
        return () => { };
    });
}
```

Есть и второе, что `Observable.Create` тихо для нас делает. При определенных обстоятельствах он использует систему планировщика Rx для вызова нашего кода через рабочую очередь вместо непосредственного вызова. Это позволяет избежать возможных взаимоблокировок в случаях, когда вы увязали вместе несколько наблюдаемых объектов. Планировщики я опишу позже в этой главе.

Эта техника подходит для холодных источников, таких как листинг 11.9. Горячие источники работают по-другому, транслируя прямые события всем

подписчикам. `Observable.Create` не обслуживает их напрямую, так как вызывает делегата, который вы передаете один раз для каждого подписчика. Но библиотеки Rx все равно еще могут пригодиться.

Rx предоставляет метод расширения `Publish` для любого `I Observable<T>`, определенный классом `Observable` в пространстве имен `System.Reactive.Linq`. Этот метод предназначен для обертки источника, чей метод подписки (т. е. делегат, передаваемый в `Observable.Create`) поддерживает только однократный запуск, но к которому вы хотите присоединить несколько подписчиков. Таким образом он предоставляет вам логику многоадресной рассылки. Строго говоря, источник, который поддерживает только одну подписку, является вырожденным, но пока вы скрываете его за `Publish`, это не имеет значения, и вы можете использовать это как способ реализации горячего источника. В листинге 11.11 показано, как создать источник, который содержит тот же функционал, что и `KeyWatcher` в листинге 11.7. Я также подключил двух подписчиков, чтобы проиллюстрировать тот факт, что он поддерживает несколько подписчиков.

Листинг 11.11. Горячий источник на основе делегатов

```
I Observable<char> singularHotSource = Observable.Create<
    (Func<I Observer<char>, IDisposable>) (obs =>
{
    while (true)
    {
        obs.OnNext(Console.ReadKey(true).KeyChar);
    }
});
```

```
I ConnectableObservable<char> keySource = singularHotSource.Publish();
```

```
keySource.Subscribe(new MySubscriber<char>());
keySource.Subscribe(new MySubscriber<char>());
```

Метод `Publish` не вызывает функцию `Subscribe` источника немедленно. Этого не происходит, даже когда вы впервые подключаете подписчика к источнику, который он возвращает. Таким образом, ко времени выполнения всего кода в листинге 11.11 цикл, который считывает нажатия клавиш, еще не будет выполняться. Я должен сообщить опубликованному источнику, если хочу, чтобы выполнение началось. Обратите внимание, что `Publish` возвращает `I ConnectableObservable<T>`. Он наследуется от `I Observable<T>` и добавляет еще один дополнительный метод, `Connect`. Этот интерфейс

представляет собой источник, который не запускается до тех пор, пока ему не скажут, и предназначен он для того, чтобы вы могли подключить всех нужных подписчиков до его фактического запуска. Вызов `Connect` источника, возвращаемого `Publish`, заставляет его подписаться на мой исходный источник, что вызывает обратный вызов метода подписки, который я передал в `Observable.Create`, и запускает мой цикл. Это приводит к тому, что метод `Connect` имеет тот же эффект, что и вызов `Run` в моем исходном листинге 11.7.



`Connect` возвращает `IDisposable`. Это дает возможность отключиться на более позднем этапе, т. е. отписаться от основного источника. (Если вы его не вызовете, подключаемый наблюдаемый объект, возвращаемый функцией `Publish`, останется подписанным на ваш источник, даже если вы удалите каждую из отдельных последующих подписок.)

Комбинация основанного на делегате `Observable.Create` и многоадресной рассылки, предлагаемой `Publish`, позволила мне в листинге 11.7 отбросить все, кроме цикла, который фактически генерирует элементы, но даже он упростился. Возможность удалить около 80% кода — это еще не все. Работать все тоже будет лучше — `Publish` позволяет Rx обрабатывать моих подписчиков и корректно справляется с затруднительными ситуациями, когда подписчики отписываются во время получения уведомлений.

Конечно, библиотеки Rx помогают не только с реализацией источников. Они также способны упростить код подписчиков.

Подписка на наблюдаемый источник с использованием делегатов

Так же как вам не нужно реализовывать `IObservable<T>`, нет необходимости и в самостоятельной реализации `IObserver<T>`. Вам не всегда придется заботиться обо всех трех методах — `KeyWatcher` из листинга 11.7 вообще никогда не вызывает методы `OnCompleted` или `OnError`, потому что работает бесконечно и не отслеживает ошибки. Даже если вам нужно предоставить все три метода, вам не обязательно писать для этого отдельный тип. Библиотеки Rx предоставляют методы расширения для упрощения подписки, определяемые классом `ObservableExtensions` в пространстве имен `System`. Большинство исходных файлов C# включают в себя директиву `«using System;»`, поэтому эти расширения обычно доступны, если в вашем проекте есть ссылка на па-

кет NuGet `System.Reactive`. Для любого `IObservable<T>` доступны несколько перегрузок метода `Subscribe`. Листинг 11.12 использует одну из них.

Листинг 11.12. Подписка без реализации `IObserver<T>`

```
var source = new KeyWatcher();
source.Subscribe(value => Console.WriteLine("Received: " + value));
source.Run();
```

Этот пример работает так же, как и листинг 11.8. Однако при таком подходе мы больше не нуждаемся в большей части кода листинга 11.2. С помощью этого метода расширения `Subscribe` Rx предоставляет нам реализацию `IObserver<T>`, а мы должны предоставить методы только для тех уведомлений, которые нам нужны.



Если вы не предоставляете обработчик исключений при использовании подписки на основе делегатов, а источник вызывает `OnError`, тогда поддерживаемый Rx `IObserver<T>` вызывает исключение, чтобы эта ошибка не осталась незамеченной. Листинг 11.5 вызывает `OnError` в блоке `catch`, где он обрабатывает исключения ввода-вывода, и, если вы подписались с использованием техники из листинга 11.12, вы обнаружите, что вызов `OnError` немедленно выдает `IOException` — то же самое исключение, которое затем выдается дважды подряд. Один раз его вызывает `StreamReader`, а затем еще раз предоставленная Rx реализация `IObserver<T>`. Поскольку к этому времени мы уже в блоке `catch` листинга 11.5 (а не в блоке `try`), этот второй выброс вызовет исключение в методе `Subscribe`, и оно либо отправится дальше по стеку, либо вызовет сбой приложения.

Перегрузка `Subscribe`, используемая в листинге 11.12, получает `Action<T>`, где `T` — тип элемента `IObservable<T>`, в данном случае `char`. Мой источник не отправляет уведомления об ошибках и не использует `OnCompleted`, чтобы указать на окончание выдачи элементов, но есть много источников, которые это делают, так что для этого имеются три перегрузки `Subscribe`. Одна для обработки ошибки принимает дополнительный делегат типа `Action<Exception>`. Другая принимает второй делегат типа `Action` (т. е. тот, который не принимает аргументов) для обработки уведомления о завершении. Третья перегрузка принимает три делегата — общий для всех поэлементный обратный вызов, а также обработчик исключений и обработчик завершения.

Есть еще одна перегрузка метода расширения `Subscribe`, которая не принимает аргументов. Она подписывается на источник, но затем ничего не делает

с элементами, которые получает. (Он будет передавать любые ошибки обратно в источник, как и другие перегрузки, которые не принимают обратный вызов для ошибки.) Это полезно, если у вас есть источник, который делает что-то важное в качестве побочного эффекта подписки, хотя, вероятно, лучше избегать такого дизайна, который требует такого поведения.

Построители последовательностей

Rx определяет несколько методов, которые создают новые последовательности с нуля, не требуя ни пользовательских типов, ни обратных вызовов. Они предназначены для определенных простых сценариев, таких как одноэлементные последовательности, пустые последовательности или конкретные шаблоны. Все это статические методы, определяемые классом `Observable`.

Empty

Метод `Observable.Empty<T>` похож на `Enumerable.Empty<T>` из `LINQ to Objects`, который я показал в главе 10: он создает пустую последовательность. (Разница, конечно, в том, что он реализует `IObservable<T>` вместо `IEnumerable<T>`.) Как и в случае с методом из `LINQ to Objects`, он нужен, когда вы работаете с API, которым требуется наблюдаемый источник, а у вас нет элементов для предоставления.

Любой наблюдатель, который подписывается на последовательность `Observable.Empty<T>`, немедленно получит вызов метода `OnCompleted`.

Never

Метод `Observable.Never<T>` создает последовательность, которая никогда ничего не делает. Она не создает элементов и, в отличие от пустой последовательности, даже не завершается. (Команда Rx думала назвать его `Infinite<T>`, чтобы подчеркнуть тот факт, что, помимо того что она никогда ничего не производит, она еще и никогда не заканчивается.) В `LINQ to Objects` аналога этому методу нет. Если вы хотите написать эквивалент `Never` на основе `IEnumerable<T>`, он навсегда заблокируется при первой попытке получить элемент. В случае `LINQ to Objects`, который основан на извлечении, это было бы совершенно бесполезно — вызывающий поток завис бы на все время существования процесса. Но в реактивной среде Rx

источники не блокируют потоки только потому, что находятся в состоянии, в котором в данный момент не производят предметы. Следовательно, `Never` не будет такой уж катастрофической идеей. Метод может пригодиться для некоторых операторов, о которых я расскажу позже и которые могут использовать `IObservable<T>` для представления длительности работы. `Never` может представлять действие, которое должно выполняться бесконечно.

Return

Метод `Observable.Return<T>` принимает один аргумент и возвращает наблюдаемую последовательность, которая немедленно выдает это единственное значение, после чего завершается. Это холодный источник — вы можете подписаться на него любое количество раз, и каждый подписчик получит одно и то же значение. В LINQ to Objects у него нет точного эквивалента, но команда Rx предоставляет библиотеку под названием «Интерактивные расширения для .NET» (сокращаемую до `Ix` и доступную в пакете NuGet `System.Interactive`), которая содержит версии на основе `IEnumerable<T>` для этого и нескольких других операторов, описанных в этой главе и присутствующих в Rx, но недоступных в LINQ to Objects.

Throw

Метод `Observable.Throw<T>` принимает один аргумент типа `Exception` и возвращает наблюдаемую последовательность, которая немедленно передает это исключение в метод `OnError` любого подписчика. Как и `Return`, это холодный источник, на который можно подписаться любое количество раз, и он будет делать одно и то же для каждого подписчика.

Range

Метод `Observable.Range` генерирует последовательность чисел. Как и метод `Enumerable.Range`, он принимает начальное число и количество. Это холодный источник, который будет выдавать весь диапазон каждому подписчику.

Repeat

Метод `Observable.Repeat<T>` получает входные данные и создает последовательность, которая снова и снова выдает эти входные данные. Входные

данные могут быть единственным значением, но могут быть и другой наблюдаемой последовательностью, и в этом случае элементы будут пересыпаться до тех пор, пока не завершится входная последовательность, после чего произойдет повторная подпись для следующей выдачи всей последовательности. (Это означает, что данные будут действительно повторяться только в том случае, если вы передадите холдовый наблюдаемый источник.)

Если вы не передадите никаких других аргументов, результирующая последовательность будет генерировать значения бесконечно, и единственный способ остановить это — отписаться. Вы также можете передать количество, указав, сколько повторов входных данных вы бы хотели получить.

Generate

Метод `Observable.Generate<TState, TResult>` умеет создавать более сложные последовательности, чем методы, которые я только что описал. Вы реализуете `Generate` с объектом или значением, представляющим собой начальное состояние генератора. Это может быть любой нужный вам тип, так как это один из аргументов метода обобщенного типа. Кроме этого, вы должны предоставить три функции: одну для проверки текущего состояния, чтобы определить, завершена ли последовательность, другую для изменения состояния при подготовке к созданию следующего элемента и третью для определения значения для выдачи в текущем состоянии. Листинг 11.13 использует все это для создания источника, который выдает случайные числа, пока общая сумма всех чисел не превысит 10 000.

Листинг 11.13. Создание элементов

```
IObservable<int> src = Observable.Generate(
    Current: 0, Total: 0, Random: new Random(),
    state => state.Total <= 10000,
    state =>
{
    int value = state.Random.Next(1000);
    return (value, state.Total + value, state.Random);
},
state => state.Current);
```

Пример всегда выдает 0 в качестве первого элемента, иллюстрируя вызов функции, которая определяет текущее значение (последняя лямбда в листинге 11.13) перед первым вызовом функции, которая выполняет перебор состояния.

Вы можете добиться того же эффекта, используя `Observable.Create` и цикл. Однако `Generate` инвертирует поток управления: вместо того чтобы держать ваш код в цикле и заставлять его сообщать Rx, когда нужно выдать следующий элемент, Rx обращается к вашим функциям для каждого следующего элемента. Это дает Rx больше гибкости по сравнению с планированием работы. Например, это позволяет `Generate` иметь перегрузки, которые добавляют в картину синхронизацию. В листинге 11.14 элементы создаются аналогичным образом, но в качестве последнего аргумента передается дополнительная функция, которая указывает Rx отложить выдачу каждого элемента на случайный срок.

Листинг 11.14. Генерация элементов с привязкой ко времени

```
IObservable<int> src = Observable.Generate(
    (Current: 0, Total: 0, Random: new Random()),
    state => state.Total < 10000,
    state =>
{
    int value = state.Random.Next(1000);
    return (value, state.Total + value, state.Random);
},
state => state.Current,
state => TimeSpan.FromMilliseconds(state.Random.Next(1000)));
```

Чтобы это работало, Rx нужна возможность запланировать работу на какой-то момент в будущем. Я объясню, как это работает в разделе «Планировщики» на стр. 517.

Запросы LINQ

Одним из главных преимуществ использования Rx является то, что он имеет реализацию LINQ, позволяющую писать запросы для обработки асинхронных потоков элементов, например событий. Листинг 11.15 иллюстрирует это. Он начинается с создания наблюдаемого источника, представляющего события `MouseMove` из элемента пользовательского интерфейса. Я расскажу об этом методе более подробно в разделе «Адаптация» на с. 650, но сейчас достаточно знать, что Rx может создать обертку для любого события .NET, превратив его в наблюдаемый источник. Каждое событие создает элемент с двумя свойствами, содержащими значения, которые обычно передаются обработчикам событий в качестве аргументов (т. е. отправитель и аргументы события).

Листинг 11.15. Фильтрация элементов с помощью запроса LINQ

```
Iobservable<EventPattern<MouseEventArgs>> mouseMoves =  
    Observable.FromEventPattern<MouseEventArgs>(  
        background, nameof(background.MouseMove));  
  
Iobservable<Point> dragPositions =  
    from move in mouseMoves  
    where Mouse.Captured == background  
    select move.EventArgs.GetPosition(background);  
  
dragPositions.Subscribe(point => { line.Points.Add(point); });
```

Выражение `where` в запросе LINQ фильтрует события, поэтому мы обрабатываем только те из них, которые были вызваны, когда конкретный элемент пользовательского интерфейса (`background`) захватил мышь. Этот конкретный пример основан на WPF, но в целом настольные приложения Windows, которым требуется поддержка перетаскивания, захватывают мышь при нажатии кнопки мыши и отпускают ее после. Это гарантирует, что захватывающий элемент получает события перемещения мыши в течение всего времени перетаскивания, даже если мышь перемещается над другими элементами пользовательского интерфейса. Обычно элементы пользовательского интерфейса получают события перемещения мыши, когда мышь находится над ними, даже если они ее не захватили. Поэтому мне нужно, чтобы выражение `where` в листинге 11.15 игнорировало эти события, оставляя только те движения мыши, которые происходят во время перетаскивания. Таким образом, чтобы код в листинге 11.15 работал, вам нужно присоединить обработчики событий, такие как показаны в листинге 11.16, к событиям `MouseDown` и `MouseUp` соответствующего элемента.

Листинг 11.16. Захват мыши

```
private void OnBackgroundMouseDown(object sender, MouseButtonEventArgs e)  
{  
    background.CaptureMouse();  
}  
  
private void OnBackgroundMouseUp(object sender, MouseButtonEventArgs e)  
{  
    if (Mouse.Captured == background)  
    {  
        background.ReleaseMouseCapture();  
    }  
}
```

Выражение `select` в листинге 11.15 работает в Rx так же, как в `LINQ to Objects` или в любом другом провайдере LINQ. Оно позволяет нам извлекать информацию из исходных элементов для использования в качестве вывода. В данном случае `mouseMoves` является наблюдаемой последовательностью объектов `EventPattern<MouseEventArgs>`, но то, что мне действительно нужно, — это наблюдаемая последовательность положений мыши. Так что выражение `select` в листинге 11.15 запрашивает позицию относительно определенного элемента пользовательского интерфейса.



Операторы не интересует, горячий ли источник. Операторы `Where` и `Select` просто передают этот аспект как есть. Каждый раз, когда вы подписываетесь на последний запрос, выдаваемый оператором `Select`, он будет подписываться на его ввод. В данном случае вход — это наблюдаемый объект, возвращаемый оператором `Where`, который, в свою очередь, будет подписан на источник, созданный путем адаптации событий перемещения мыши. Если вы подпишетесь во второй раз, вы получите вторую цепочку подписок. Горячий источник событий будет транслировать каждое событие в обе цепочки, из-за чего каждый элемент будет проходить процесс фильтрации и проецирования дважды. Поэтому следует помнить, что подключение нескольких подписчиков к сложному запросу горячего источника будет работать, но может повлечь за собой ненужные расходы. Если вам это все же нужно, стоит подумать о вызове метода запроса `Publish`, который, как вы уже видели, способен создать единственную подписку на свой вход, а затем многоадресно передать каждый элемент всем подписчикам.

Результатом такого запроса является то, что `dragPositions` ссылается на наблюдаемую последовательность значений `Point`, которая будет сообщать о каждом изменении положения мыши, происходящем, когда определенный элемент пользовательского интерфейса в моем приложении захватывает мышь. Это горячий источник, потому что он выдает то, что происходит в настоящем, а именно ввод с помощью мыши. Операторы фильтрации и проекции LINQ не изменяют природу источника, поэтому, если вы примените их к горячему источнику, результирующий запрос тоже будет горячим, а если источник холодный, отфильтрованный результат будет таким же.

Последняя строка листинга 11.15 подписывается на отфильтрованный и спроектированный источник и добавляет каждое значение `Point`, которое он выдает, к коллекции `Points` другого элемента пользовательского интерфейса, называемого `line`. Это элемент `Polyline`, который здесь не показан, так что

в результате вы можете с помощью мыши оставлять росчерки в окне приложения. (Если вы занимались разработкой для Windows достаточно долго, то можете вспомнить примеры Scribble – эффект здесь практически тот же.)

Rx предоставляет большинство стандартных операторов запросов, описанных в главе 10. Большинство из них работают в Rx точно так же, как и в других реализациях LINQ¹. Тем не менее некоторые из них на первый взгляд работают довольно удивительно, что я опишу в следующих нескольких разделах.

Операторы группировки

Стандартный оператор группировки `GroupBy` создает последовательность последовательностей. В LINQ to Objects он возвращает `IEnumerable<IGrouping< TKey , TSource >>` и, как вы видели в главе 10, сам `IGrouping< TKey , TSource >` наследуется от `IEnumerable< TSource >`. `GroupJoin` имеет схожую идею: хотя он возвращает простой `IEnumerable< T >`, а `T` в нем является результатом функции проекции, которая передается последовательности в качестве входного потока. Таким образом, вы в любом случае получаете то, что логически представляет собой последовательность последовательностей.

В среде Rx группировка создает наблюдаемую последовательность наблюдаемых последовательностей. Это совершенно непротиворечиво, но может немного удивлять, потому что Rx добавляет временной аспект: наблюдаемый источник, который представляет все группы, производит новый элемент (новый наблюдаемый источник) в тот момент, когда он обнаруживает каждую новую группу. Листинг 11.17 иллюстрирует это, наблюдая за изменениями в файловой системе, и затем формирует из них группы на основе папки, в которой они произошли. Для каждой группы мы получаем `IGroupedObservable< TKey , TSource >`, который является эквивалентом `IGrouping< TKey , TSource >` в Rx.

Листинг 11.17. Группировка событий

```
string path = Environment.GetFolderPath(
    Environment.SpecialFolder.MyDocuments);
var w = new FileSystemWatcher(path);
```

¹ В нем нет операторов `OrderBy` и `ThenBy`, так как они не имеют большого смысла в реактивном окружении. Они не способны выдавать никаких элементов, пока не получат все свои входные элементы.

```

IObservable<EventPattern<FileSystemEventArgs>> changes =
    Observable.FromEventPattern<FileSystemEventHandler,
    FileSystemEventArgs>(
        h => w.Changed += h, h => w.Changed -= h);
w.IncludeSubdirectories = true;
w.EnableRaisingEvents = true;

IObservable<IGroupedObservable<string, string>> folders =
    from change in changes
    group Path.GetFileName(change.EventArgs.FullPath)
        by Path.GetDirectoryName(change.EventArgs.FullPath);

folders.Subscribe(f =>
{
    Console.WriteLine("New folder ({0})", f.Key);
    f.Subscribe(file =>
        Console.WriteLine("File changed in folder {0}, {1}", f.Key, file));
});

```

Лямбда, которая подписывается на источник группировки, `folders`, подписывается на каждую группу, которую производит источник. Количество папок, в которых могут происходить события, бесконечно, поскольку новые могут добавляться во время работы программы. Таким образом, наблюдаемый источник `folders` будет генерировать новый наблюдаемый источник каждый раз, когда обнаружит изменение в папке, которую он раньше не видел, что показано на рис. 11.2.

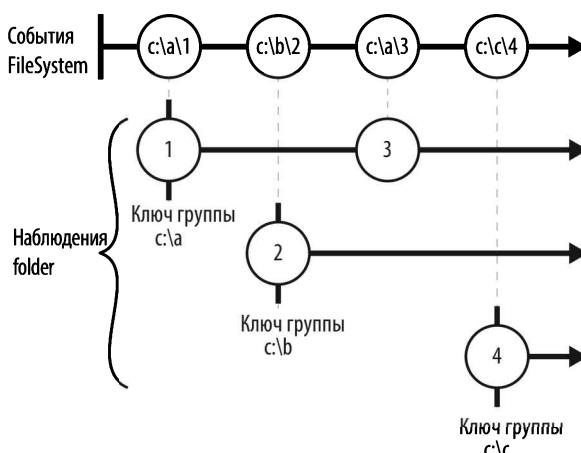


Рис. 11.2. Разбиение `IObservable<T>` на группы

Обратите внимание, что создание новой группы не означает, что все предыдущие группы завершают работу. Это поведение отличается от того, как работает группировка в LINQ to Objects. Когда вы выполняете группирующий запрос в `IEnumerable<T>`, то по мере создания группы вы можете полностью перебрать ее содержимое, прежде чем переходить к следующей. Но вы не можете сделать это в Rx, потому что каждая группа представлена как наблюдаемая и наблюдаемые объекты не завершены, пока они вам об этом не сообщат. Так что каждая подписка на группу остается активной. В листинге 11.17 вполне возможно, что папка, для которой группа уже была создана, будет неактивна в течение длительного времени, тогда как в других папках будет происходить активность, но активизируется позже. Вообще, операторы группировки Rx должны быть готовы к тому, что подобное может произойти с любым источником.

Операторы объединения

Rx предоставляет стандартные операторы `Join` и `GroupJoin`. Однако они работают немного иначе, чем обработка объединений в LINQ to Objects или большинстве провайдеров LINQ баз данных. В этих средах элементы из двух входных наборов обычно объединяются на основе общего значения. В случае баз данных очень распространенным примером объединения двух таблиц служит объединение строк, в которых столбец внешнего ключа в строке из одной таблицы имеет то же значение, что и столбец первичного ключа в строке из другой таблицы. Однако объединение в Rx основано не на значениях. Вместо этого элементы объединяются, если они являются одновременными, т. е. их время работы пересекается.

Но подождите минутку! Что это вообще такое, время работы элемента? Rx имеет дело с мгновенными событиями; создание элемента, сообщение об ошибке и завершение потока — все это происходит в определенный момент. Поэтому операторы объединения используют следующее соглашение: для каждого исходного элемента вы можете предоставить функцию, которая возвращает `IObservable<T>`. Время работы этого исходного элемента начинается, когда элемент создается, и заканчивается, когда соответствующий `IObservable<T>` впервые реагирует (т. е. он либо завершает работу, либо выдает элемент или ошибку). Рисунок 11.3 иллюстрирует эту идею. Вверху расположены наблюдаемый источник, под которым находится ряд других источников, которые и определяют время работы каждого элемента. Вни-

зу я показал продолжительность, которую наблюдаемые объекты каждого элемента устанавливают для своих исходных элементов.

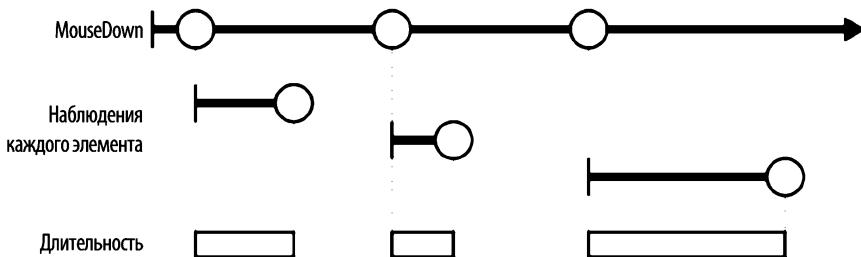


Рис. 11.3. Определение времени работы каждого исходного элемента с помощью `IObservable<T>`

Хотя вы можете использовать новый `IObservable<T>` для каждого элемента источника, как это показано на рис. 11.3, это не является требованием. Вполне допустимо каждый раз использовать один и тот же источник. Например, если вы примените группирующий оператор к `IObservable<T>`, представляющий собой поток событий `MouseDown`, после чего используете другой `IObservable<T>`, представляющий собой поток событий `MouseUp`, для определения времени работы каждого элемента, то это заставит Rx думать, что «время работы» каждого события `MouseDown` длится до следующего события `MouseUp`. Эта схема отражена на рис. 11.4 — фактическая продолжительность каждого события `MouseDown`, показанного внизу, определяется парой событий: `MouseDown` и `MouseUp`.

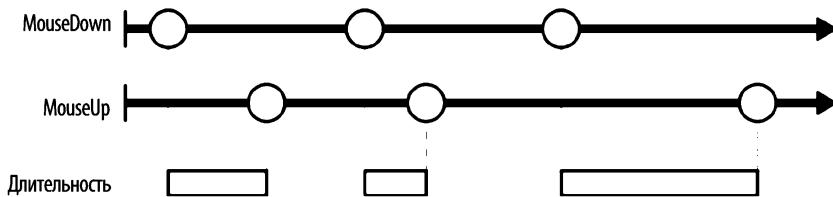


Рис. 11.4. Определение времени работы с помощью пары потоков событий

Источник способен определить даже собственное время работы. Например, если у вас есть наблюдаемый источник, представляющий собой события `MouseDown`, вам может понадобиться, чтобы время работы каждого элемента заканчивалось с началом времени работы следующего. Это будет означать,

что время работы элементов непрерывно — после получения первого элемента всегда будет существовать ровно один текущий элемент, и он будет последним полученным элементом. Это проиллюстрировано на рис. 11.5.

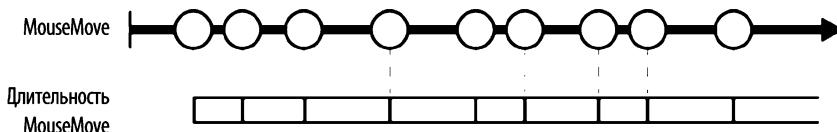


Рис. 11.5. Смежное время работы элемента

Время работы элементов может пересекаться. Если нужно, вы можете предоставить определяющий время работы `IObservable<T>`, который укажет, что время работы входного элемента заканчивается через некоторое время после начала времени работы следующего.

Теперь, когда мы знаем, как Rx определяет время работы элемента для целей объединения, стоит разобраться, как эта информация используется. Как вы помните, операторы объединения объединяют два входа. (Источники, определяющие время работы, не считаются входными.) Rx считает, что пара элементов из двух входных потоков связана, если их время работы пересекается. Способ представления связанных элементов в выводе зависит от того, используете вы оператор `Join` или `GroupJoin`. Выходные данные оператора `Join` представляют собой поток, содержащий один элемент для каждой пары связанных элементов. (Вы предоставляете функцию проекции, которая будет передаваться каждой паре элементов, и именно вам решать, что с ними делать. Эта функция позволяет определить тип выходного элемента объединенного потока.) На рис. 11.6 показаны два входных потока, которые основаны на событиях и соответствующем времени работы. Они аналогичны источникам на рис. 11.4 и 11.5, но я добавил буквы и цифры, чтобы было легче ссылаться на каждый из их элементов. В нижней части диаграммы находится наблюдаемый объект, который будет создан оператором `Join` для этих двух потоков.

Как вы видите, в любом месте, где время работы двух элементов из входных потоков пересекается, мы получаем выходной элемент, объединяющий два входа. Если перекрывающиеся элементы запускались в разное время (что обычно и происходит), элемент вывода создается всякий раз, когда запускается последний из них. Событие `MouseDown` А начинается до со-

бытия `MouseMove` 1, поэтому результирующий вывод `A1` возникает там, где начинается пересечение (т. е. когда происходит событие `MouseMove` 1). Но событие 3 происходит до события `B`, поэтому объединенный выход `B3` возникает при начале `B`.

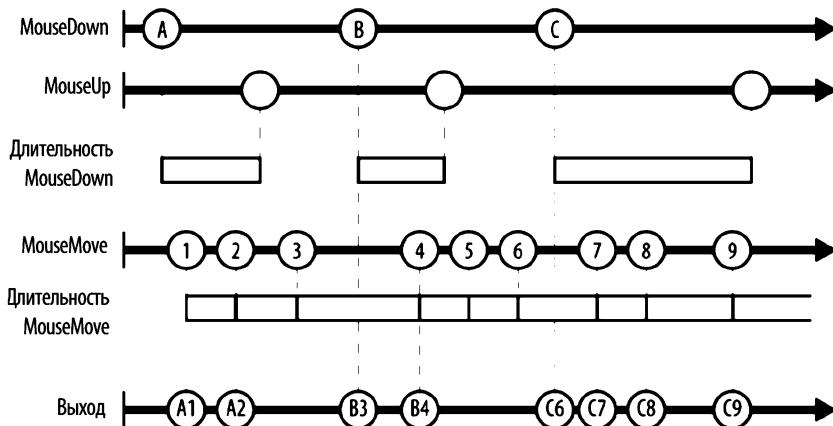


Рис. 11.6. Оператор Join

Время работы события 5 не пересекается со временем работы любых элементов `MouseDown`, поэтому мы не видим никаких соответствующих элементов в выходном потоке. С другой стороны, вполне возможно, чтобы событие `MouseMove` появлялось в нескольких выходных элементах (как это делает каждое событие `MouseDown`). Если бы не было события 3, событие 2 имело бы время работы, которое начиналось бы внутри А и заканчивалось внутри В, так что вместе с `A2`, показанным на рис. 11.6, там бы имелось событие `B2`, начинающееся одновременно с В.

В листинге 11.18 показан код, который с помощью выражения запроса выполняет объединение, показанное на рис. 11.6. Как вы помните из главы 10, компилятор превращает выражения запроса в серию вызовов методов, а листинг 11.19 показывает основанный на методах эквивалент запроса из листинга 11.18.

Листинг 11.18. Выражение запроса с объединением

```
IEnumerable<EventPattern<MouseEventArgs>> downs =
    Observable.FromEventPattern<MouseEventArgs>(
        background, nameof(background.MouseDown));
```

```
I0bservable<EventPattern<MouseEventArgs>> ups =
    Observable.FromEventPattern<MouseEventArgs>(
        background, nameof(background.MouseUp));
I0bservable<EventPattern<MouseEventArgs>> allMoves =
    Observable.FromEventPattern<MouseEventArgs>(
        background, nameof(background.MouseMove));

I0bservable<Point> dragPositions =
    from down in downs
    join move in allMoves
        on ups equals allMoves
    select move.EventArgs.GetPosition(background);
```

Листинг 11.19. Join в коде

```
I0bservable<Point> dragPositions =
    downs.Join( allMoves,
        down => ups,
        move => allMoves,
        (down, move) => move.EventArgs.GetPosition(background));
```

Мы можем использовать наблюдаемый источник `dragPositions`, созданный любым из этих примеров, для замены того, что показан в листинге 11.15. Нам больше не нужна фильтрация на основе того, захватил ли элемент `background` мышь, потому что Rx теперь предоставляет нам только события перемещения, время работы которых пересекается с продолжительностью события нажатия мыши. Любые движения, которые происходят между нажатиями мыши, будут либо игнорироваться, либо, если они произошли непосредственно перед нажатием, мы получим их позицию в момент нажатия кнопки мыши.

`GroupJoin` объединяет элементы аналогичным образом, но вместо того, чтобы создавать один наблюдаемый результат, он создает наблюдаемый источник наблюдаемых источников. Для данного примера это будет означать, что его выход будет содержать новый наблюдаемый источник для каждого входа `MouseDown`. Он будет состоять из всех пар, содержащих этот вход, и будет иметь ту же продолжительность, что и данный вход. На рис. 11.7 показан этот оператор в действии, с теми же входными событиями, что и на рис. 11.6. Я обозначил вертикальными линиями завершения выходных последовательностей, чтобы было ясно, когда они будут вызывать методы `OnComplete` своих наблюдателей. Начало и конец работы этих наблюдаемых источников точно совпадают с временем работы соответствующего входа, поэтому они часто заканчивают через некоторое время после выдачи своего конечного элемента вывода.

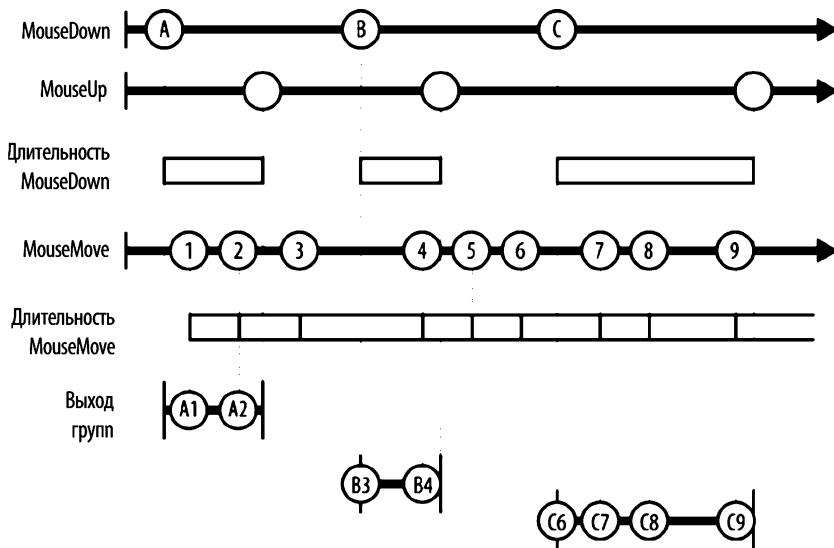


Рис. 11.7. Оператор GroupJoin

Если говорить о LINQ в целом, оператор `GroupJoin` может создавать пустые группы, поэтому, в отличие от оператора `Join`, для каждого первого элемента входа будет создан один выход, даже если в другом потоке нет соответствующих элементов. `GroupJoin` библиотеки Rx работает так же, но добавляет фактор времени. Каждая группа выхода начинается в один и тот же момент, когда происходит соответствующее событие входа (в данном примере `MouseDown`), и заканчивается, когда считается, что это событие завершилось (в данном случае при следующем `MouseUp`); если за это время не было никаких движений, этот наблюдаемый объект не будет генерировать никаких элементов. Поскольку длительность событий перемещения здесь непрерывна, это может произойти только до получения первого движения. Но в соединениях, где элементы второго входа имеют непрерывную длительность, пустые группы более вероятны.

В контексте моего примера приложения, которое позволяет пользователю рисовать в окне с помощью мыши, этот групповой вывод достаточно полезен, потому что представляет каждое отдельное перетаскивание как отдельный объект. Это означает, что я могу создать новую линию для каждого перетаскивания, а не добавлять точки на одну и ту же более длинную линию. С кодом листинга 11.15 каждая новая операция перетаскивания будет рисовать линию от того места, где закончилось предыдущее перетаскивание, до

нового местоположения, делая невозможным рисование отдельных фигур. Но сгруппированный вывод облегчает разделение. Листинг 11.20 подписывается на сгруппированный вывод, и для каждой новой группы (новой операции перетаскивания) он создает новую `Polyline` для отображения каракулей, после чего подписывается на элементы в группе для заполнения этой отдельной линии.

Листинг 11.20. Добавление новой линии для каждой операции перетаскивания

```
var dragPointSets = from mouseDown in downs
    join move in allMoves
    on ups equals allMoves into m
    select m.Select(e => e.EventArgs.GetPosition(background));

dragPointSets.Subscribe(dragPoints =>
{
    var currentLine = new Polyline { Stroke = Brushes.Black,
        StrokeThickness = 2 };
    background.Children.Add(currentLine);

    dragPoints.Subscribe(point =>
    {
        currentLine.Points.Add(point);
    });
});
```

Чтобы было понятно, все это работает в режиме реального времени даже с оператором объединения, так что все это — горячие источники. `IEnumerable<IObservable<Point>>`, возвращаемый `GroupJoin` в листинге 11.20, создаст новую группу в момент нажатия кнопки мыши. `IObservable<Point>` из этой группы немедленно создаст новый `Point` для каждого события `MouseMove`. В результате пользователь видит, как появившаяся линия увеличивается при перемещении мыши.

Оператор `SelectMany`

В главе 10 мы видели, что оператор `SelectMany` по сути сводит коллекцию коллекций в одну. Этот оператор используется, когда выражение запроса имеет несколько выражений `from`, а в случае LINQ to Objects его работа аналогична работе вложенных циклов `foreach`. В Rx у него все еще присутствует этот эффект выравнивания — он позволяет взять наблюдаемый источник, где каждый элемент, который он производит, тоже является наблюдаемым

источником (или может использоваться для его генерации), и в результате работы оператора `SelectMany` получить одну наблюдаемую последовательность, которая содержит все элементы дочерних источников. Однако, как и в случае с группировкой, результат несколько менее аккуратен, чем в случае LINQ to Objects. Активная природа библиотеки Rx и ее потенциал асинхронной работы позволяет всем задействованным наблюдаемым источникам выдавать новые элементы одновременно, включая исходный источник, который используется в качестве источника вложенных источников. (Оператор по-прежнему гарантирует, что за один раз будет доставлено только одно событие — когда он вызывает `OnNext`, он ожидает его возврата, прежде чем сделать следующий вызов. Возможный беспорядок может возникнуть лишь тогда, когда будет перепутан порядок, в котором доставляются события.)

Когда вы используете LINQ to Objects для перебора ступенчатого массива, все происходит в прямом порядке. Он извлекает первый вложенный массив и перебирает все его элементы, прежде чем перейти к следующему вложенному массиву и перебрать его, и т. д. Но такое аккуратное выравнивание происходит только потому, что в случае `IEnumerable<T>` потребитель элементов сам решает, когда и какие элементы извлекать. С помощью Rx подписчики получают элементы тогда, когда их предоставляют источники.

Несмотря на свою произвольность, поведение достаточно простое: поток вывода, производимый `SelectMany`, просто предоставляет элементы по мере того, как их предоставляют источники.

Агрегация и другие операторы с единичным значением

Некоторые из стандартных операторов LINQ сводят всю последовательность значений к одному. К ним относятся операторы агрегирования, такие как `Min`, `Sum` и `Aggregate`, кванторы `Any` и `All`, а также оператор `Count`. Также в их число входят некоторые выбранные операторы, такие как `ElementAt`. Они доступны в Rx, но, в отличие от большинства реализаций LINQ, реализации Rx не возвращают простые единичные значения. Все они возвращают `IObservable <T>` точно так же, как операторы, которые в виде выхода выдают последовательности.

Каждый из этих операторов по-прежнему выдает единственное значение, но все они представляют это значение в виде наблюдаемого источника. Причина в том, что, в отличие от LINQ to Objects, Rx не может перебрать

свои входные данные, чтобы вычислить совокупное значение или найти выбранный элемент. Это контролируется источником, поэтому Rx-версии этих операторов должны ожидать, пока источник предоставит свои значения. Как и все операторы, операторы с единичным значением должны оставаться реактивными, а не проактивными. Операторы, которым требуется каждое значение, например `Average`, не смогут выдать свой результат, пока источник не сообщит, что он закончил работу. Даже оператор, которому не нужно ждать самого конца ввода, такой как `FirstAsync` или `ElementAt`, по-прежнему не может ничего сделать, пока источник не решит предоставить значение, которое этот оператор ожидает. Как только такой оператор получает возможность предоставить значение, он это делает, после чего завершается.



Операторы `First`, `Last`, `FirstOrDefault`, `LastOrDefault`, `Single` и `SingleOrDefault` должны бы работать одинаково, но по историческим причинам не работают. Появившись в первой версии Rx, они возвращали отдельные значения, которые не были обернуты в `IEnumerable<T>`. Это означало, что они блокировались, пока источник не предоставлял то, что им нужно. Это не очень вписывается в модель активных источников и способно привести к возникновению тупиковой ситуации, поэтому теперь они устарели, а вместо них появились новые асинхронные версии, которые работают так же, как и другие операторы Rx с единичным значением. Все они получили к своим начальным именам общий префикс `Async` (например, `FirstAsync`, `LastAsync` и т. д.).

Операторы `ToDictionary`, `ToLookup` и `ToList` работают аналогичным образом. Хотя все они выдают содержимое источника полностью, они делают это в виде единственного объекта вывода, который выглядит как наблюдаемый источник из одного элемента.

Если вам действительно требуется ожидать значения какого-либо из этих элементов, вы можете использовать оператор `Wait`, специфичный для Rx нестандартный оператор, доступный для любого `IEnumerable<T>`. Этот блокирующий оператор ожидает завершения работы источника, после чего возвращает последний элемент, так что поведение в стиле «сидеть и ждать» устаревших операторов `First`, `Last` и др. по-прежнему доступно, просто больше не используется по умолчанию. В качестве альтернативы вы можете использовать функции асинхронного языка C# — передать ключевому слову `await` наблюдаемый источник. Логически он делает то же самое, что и `Wait`, но с продуктивным неблокирующими асинхронным ожиданием, описанным в главе 17.

Оператор `Concat`

Оператор `Concat` в Rx использует ту же концепцию, что и в других реализациях LINQ: он объединяет две входные последовательности, чтобы создать последовательность, которая будет выдавать каждый элемент первого входа, а затем каждый элемент второго. (Rx заходит дальше других провайдеров LINQ и может принять и объединить коллекцию входов.) Это полезно, только если первый поток в конечном итоге завершается. Конечно, это верно и для LINQ to Objects, но бесконечные источники более распространены в Rx. Также имейте в виду, что этот оператор не подписывается на второй поток, пока не завершится первый. Это связано с тем, что холодные потоки обычно начинают создавать элементы при подписке, и оператор `Concat` не хочет буферизовать элементы второго источника, пока ожидает завершения первого. Это означает, что при использовании с горячими источниками `Concat` может выдавать недетерминированные результаты. (Если вам нужен наблюдаемый источник, который содержит все элементы двух горячих источников, используйте `Merge`, который я скоро опишу.)

Библиотеке Rx недостаточно просто предоставить стандартные операторы LINQ. Дополнительно она определяет множество собственных операторов.

Операторы запроса из Rx

Одна из основных целей Rx — упрощение работы с несколькими потенциально независимыми наблюдаемыми источниками, которые выдают элементы асинхронно. Дизайнеры Rx иногда упоминают «дирижирование и синхронизацию», имея в виду, что в вашей системе может одновременно происходить множество событий, но вам необходимо достичь определенной согласованности в том, как ваше приложение на них реагирует. Многие операторы Rx разработаны именно с этой целью.



Не все в этом разделе обусловлено уникальностью требований Rx. Некоторые из нестандартных операторов Rx (например, `Scan`) можно было бы применить и в других провайдерах LINQ. Версии многих из них доступны для `IEnumerable<T>` в «Интерактивных расширениях для .NET» (`Ix`), которые, как упоминалось ранее, можно найти в пакете `NuGet System.Interactive`.

Rx имеет настолько большой выбор операторов, что рассказ о них увеличил бы главу раза в четыре, а она уже и так не маленькая. Поскольку это не книга об Rx и некоторые операторы очень специализированы, я просто упомяну ряд самых полезных. Рекомендую обратиться к документации по Rx (или исходному коду по адресу <https://github.com/dotnet/reactive>), где можно найти полный и на удивление исчерпывающий набор предоставляемых операторов.

Merge

Оператор `Merge` объединяет все элементы из двух или более наблюдаемых последовательностей в одну. Это можно использовать, чтобы избавиться от проблемы, которая возникает в листингах 11.15, 11.18 и 11.20. Все они обрабатывают ввод с помощью мыши, и, если вы хорошо знакомы с программированием пользовательского интерфейса Windows, вам известно, что вы не обязательно получите уведомление о перемещении мыши, соответствующее точкам, в которых была нажата и отпущена кнопка мыши. Уведомления об этих событиях кнопок содержат информацию о расположении мыши, поэтому Windows не видит необходимости отправлять отдельное сообщение о перемещении мыши с указанием этих местоположений, поскольку попросту отправит вам одну и ту же информацию дважды. Хоть это совершенно логичное поведение, оно все же раздражает². Начальные и конечные местоположения не содержатся в наблюдаемом источнике, которым в этих примерах представлены положения мыши. Это можно исправить, объединив позиции всех трех событий. Листинг 11.21 показывает, как можно исправить листинг 11.15.

Листинг 11.21. Слияние наблюдаемых источников

```
IObserveable<EventPattern<MouseEventArgs>> downs =
    Observeable.FromEventPattern<MouseEventArgs>(
        background, nameof(background.MouseDown));
IObserveable<EventPattern<MouseEventArgs>> ups =
    Observeable.FromEventPattern<MouseEventArgs>(
        background, nameof(background.MouseUp));
IObserveable<EventPattern<MouseEventArgs>> allMoves =
    Observeable.FromEventPattern<MouseEventArgs>(
        background, nameof(background.MouseMove));
```

² Как и некоторые разработчики.

```
I0bservable<EventPattern<MouseEventArgs>> dragMoves =
    from move in allMoves
    where Mouse.Captured == background
    select move;

I0bservable<EventPattern<MouseEventArgs>> allDragPositionEvents =
    Observable.Merge(downs, ups, dragMoves);

I0bservable<Point> dragPositions =
    from move in allDragPositionEvents
    select move.EventArgs.GetPosition(background);
```

Я создал три наблюдаемых источника для представления трех событий соответственно: `MouseDown`, `MouseUp` и `MouseMove`. Поскольку все три должны использовать одну и ту же проекцию (выражение `select`), но только одна должна фильтровать события, я немного изменил структуру. Только движения мыши нуждаются в фильтрации, поэтому для них я написал отдельный запрос. Затем я использовал метод `Observable.Merge`, чтобы объединить все три потока событий в один.



`Merge` доступен как в виде метода расширения, так и в виде статического метода без расширения. Если вы используете методы расширения для одного наблюдаемого источника, то доступными перегрузками `Merge` будут только те, которые объединяют его с другим источником (с необязательным указанием планировщика). В данном случае у меня было три источника, поэтому я использовал вариант без расширения. Однако если у вас есть выражение, которое является либо перечислением наблюдаемых источников, либо наблюдаемым источником наблюдаемых источников, вы обнаружите, что для них также существуют методы расширения `Merge`. Так что я мог бы написать `new[] {downs, ups, dragMoves}.Merge ()`.

Моя переменная `allDragPositionEvents` ссылается на один наблюдаемый поток, который будет сообщать обо всех нужных мне движениях мыши. Наконец, я пропускаю этот результат через проекцию, чтобы извлечь положение мыши для каждого элемента. Опять же, результат — это горячий поток. Как и прежде, он будет выдавать положение всякий раз, когда мышь перемещается, а мышь захвачена элементом `background`, но также будет выдавать позицию каждый раз, когда происходит событие `MouseDown` или `MouseUp`. Чтобы поддерживать свой пользовательский интерфейс в актуальном состоянии, я мог бы подписаться на эти события с помощью вызова,

показанного в последней строке листинга 11.15, и на этот раз я бы не пропустил начальную и конечную позиции.

В примере, который я только что показал, все источники бесконечны, но это не всегда так. Что должен делать объединенный наблюдаемый источник, когда один из его входов остановился? Если поток останавливается из-за ошибки, эта ошибка будет передана объединенному наблюдаемому объекту и в этот момент он будет завершен — наблюдаемому источнику не разрешается продолжать выдавать элементы после сообщения об ошибке. Однако, хотя вход и может в одностороннем порядке завершить вывод с ошибкой, если входы завершаются нормально, объединенный наблюдаемый объект не завершается до завершения всех своих входов.

Операторы окна

Rx определяет два оператора, `Buffer` и `Window`: оба производят наблюдаемый вывод, где каждый элемент основан на нескольких смежных элементах источника. (Кстати, название `Window` не имеет ничего общего с пользовательским интерфейсом.) На рис. 11.8 показаны три способа использования оператора `Buffer`. Я пронумеровал кружки, представляющие элементы на входе, а ниже расположил еще один ряд, представляющий элементы, которые появятся из наблюдаемого источника, созданного `Buffer`. Линии и числа в них указывают, какие входные элементы связаны с каждым выходным элементом. `Window` работает очень похоже, как вы вскоре увидите.

В первом случае я передал аргументы `(2, 2)`, указывающие, что я хочу, чтобы каждый элемент вывода соответствовал двум элементам ввода, и что мне нужно, чтобы новый буфер начинал работу на каждом втором элементе ввода. Это может выглядеть как два разных способа сказать одно и то же до тех пор, пока вы не посмотрите на второй пример на рис. 11.8. В нем аргументы `(3, 2)` указывают на то, что каждый элемент вывода соответствует трем элементам ввода, но я по-прежнему хочу, чтобы буферы начинались на каждом втором входе. Это означает, что каждое *окно* — набор элементов из входных данных, использующийся для создания выходного элемента, — пересекается с соседними. Это будет происходить всякий раз, когда второй аргумент, `skip`, меньше размера окна. Окно первого элемента вывода содержит первый, второй и третий входные элементы. Окно второго вывода содержит третий, четвертый и пятый, поэтому третий элемент появляется в обоих.

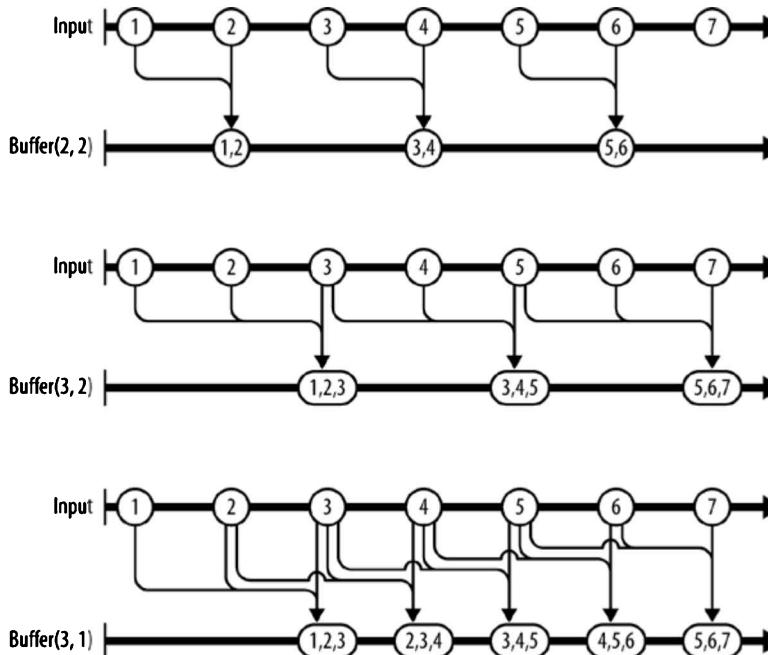


Рис. 11.8. «Сдвиг окон» с помощью оператора Buffer

В последнем примере показан размер окна, равный трем, но на этот раз я попросил, чтобы шаг был равен единице, поэтому в этом случае окно передвигается по одному элементу за раз, но включает в себя три элемента из источника. Можно указать пропуск, который больше размера окна, и в этом случае элементы ввода, попавшие между окнами, будут попросту игнорироваться.

Операторы `Buffer` и `Window` способствуют появлению задержек. Во втором и третьем случаях размер окна, равный трем, означает, что наблюдаемый входной объект должен выдать третье значение, прежде чем окно будет предоставлено в качестве элемента вывода. В случае с `Buffer` это всегда означает задержку на размер окна, но, как вы увидите позже, с использованием оператора `Window` каждое окно можно задействовать до того, как оно заполнится.

Разница между операторами `Buffer` и `Window` заключается в том, как они представляют элементы окна. `Buffer` из них самый прямолинейный. Он предоставляет `IEnumerable<IList<T>>`, где `T` — тип элемента ввода. Другими

словами, если вы подпишетесь на выходные данные `Buffer`, для каждого созданного окна подписчику будет передан список, содержащий все элементы в окне. Листинг 11.22 использует это для получения сглаженной версии местоположений мыши из листинга 11.15.

Листинг 11.22. Сглаживание ввода с помощью `Buffer`

```
IObservable<Point> smoothed = from points in dragPositions.Buffer(5, 2)
    let x = points.Average(p => p.X)
    let y = points.Average(p => p.Y)
    select new Point(x, y);
```

В первой строке этого запроса указано, что я хочу получать группы из пяти последовательных расположений мыши и мне нужна одна группа на каждый второй ввод. Оставшаяся часть запроса вычисляет среднюю позицию мыши в окне и выдает ее в качестве элемента вывода. На рис. 11.9 показан результат. Верхняя кривая является результатом использования необработанных позиций мыши. Расположенная же непосредственно под ней использует сглаженные точки, сгенерированные листингом 11.22 из тех же входных данных. Как вы можете видеть, верхняя линия довольно угловатая, но нижняя линия сглаживает множество выступов.

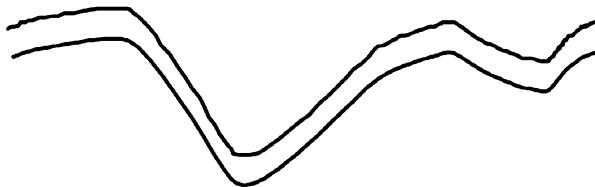


Рис. 11.9. Сглаживание в действии

Листинг 11.22 использует смесь LINQ to Objects и реализации LINQ библиотекой Rx. Само выражение запроса использует Rx, но переменная диапазона, `points`, имеет тип `IList<Point>` (потому что в этом примере `Buffer` возвращает `IObservable<IList<Point>>`).

Таким образом, вложенные запросы, которые вызывают оператор `Average` для `points`, получат реализацию LINQ to Objects.

Если входные данные оператора `Buffer` горячие, то в результате будет получен горячий наблюдаемый источник. Таким образом, вы можете подписаться на наблюдаемый источник переменной `smoothed` из листинга 11.22 с помощью

кода, аналогичного заключительной строке листинга 11.15, и в результате при перетаскивании мыши будет в реальном времени показываться сглаженная линия. Как уже говорилось, появится небольшая задержка — код указывает шаг, равный двум, поэтому он будет обновлять экран только для каждого второго события мыши. Усреднение по последним пяти точкам, в свою очередь, ведет к увеличению разрыва между указателем мыши и концом линии. С такими параметрами расхождение достаточно мало, чтобы не отвлекать, но при более агрессивном сглаживании оно может начать раздражать.

Оператор `Window` очень похож на оператор `Buffer`, но вместо того, чтобы представлять каждое окно в виде `IList<T>`, он предоставляет `IEnumerable<T>`. Если применить `Window` к `dragPositions` в листинге 11.22, результатом будет `IEnumerable<IObservable<Point>>`. Рисунок 11.10 показывает, как оператор `Window` будет работать в последнем сценарии на рис. 11.8, и, как вы можете видеть, он способен отдавать каждое окно раньше. Ему не требуется ждать, пока все элементы окна станут доступными; вместо предоставления полностью заполненного списка, содержащего окно, каждый элемент вывода является `IObservable<T>` и выдает элементы окна по мере их доступности. Каждый наблюдаемый источник, созданный `Window`, завершается сразу после предоставления последнего элемента (т. е. в тот же момент, когда `Buffer` выдал бы все окно). Таким образом, если ваша обработка зависит от наличия окна целиком, `Window` не поможет вам получить его быстрее, потому что в конечном итоге зависит от скорости поступления элементов ввода, но предоставлять значения начнет раньше.

Время запуска — это та особенность наблюдаемых источников, созданных `Window` в этом примере, которая способна удивить. Принимая во внимание, что они завершаются сразу же после выдачи последнего элемента, они не начинают работу непосредственно перед выдачей первого. Наблюдаемый источник, представляющий самое первое окно, начинает работу сразу же — вы получите его, как только подпишетесь на наблюдаемый источник из наблюдаемых источников, который возвращает оператор. Таким образом, первое окно будет доступно немедленно, даже если ввод оператора `Window` еще ничего не сделал. После этого каждое новое окно запускается только тогда, когда получены все элементы ввода, которые следовало пропустить. В этом примере я пропускаю один элемент, поэтому второе окно начинается после того, как ввод выдал один элемент, третье — после двух, и т. д.

Как вы увидите позже в этом разделе, а также в разделе «Операции для работы со временем» на с. 658, `Window` и `Buffer` используют и другие способы

для определения момента, когда каждое окно начинает и прекращает работу. Общая закономерность состоит в том, что как только оператор `Window` достигает точки, где новый элемент из источника попадет уже в новое окно, он создает это окно, предвосхищая первый элемент окна, а не ожидая его появления.

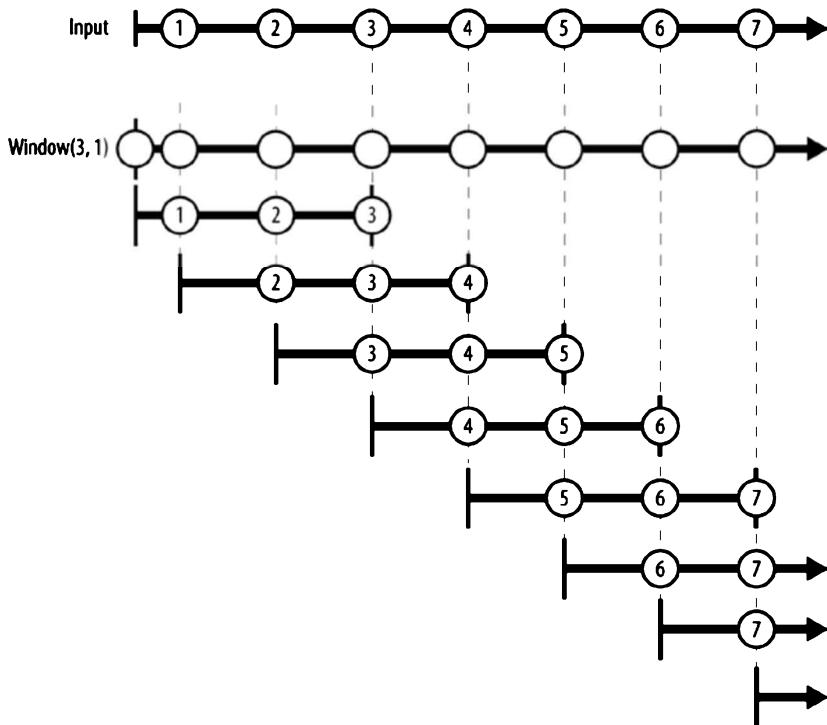


Рис. 11.10. Оператор `Window`



Если ввод завершен, все открытые в настоящее время окна также окажутся завершены. Это означает, что нам могут встретиться пустые окна. (В действительности с шагом в один элемент вы гарантированно получите одно пустое окно, если источник завершил работу.) На рис. 11.10 одно окно справа внизу запустилось, но еще не выдало ни одного элемента. Если ввод завершен без выдачи каких-либо новых элементов, три наблюдаемых источника, которые все еще находятся в работе, тоже будут завершены, включая последний, который еще вообще ничего не выдал.

Поскольку `Window` доставляет элементы в окна, как только их предоставляет источник, он позволяет вам начать обработку быстрее, чем `Buffer`, что, возможно, улучшит общую отзывчивость. Недостатком `Window` является то, что он зачастую оказывается более сложным — ваши подписчики начнут получать выходные значения до того, как станут доступны все элементы соответствующего окна ввода. Принимая во внимание, что `Buffer` предоставляет вам список, который вы можете просмотреть в любое удобное время, при использовании `Window` вы вынуждены продолжать работать с последовательностями Rx, которые выдают элементы по мере готовности. Чтобы выполнить такое же сглаживание, как в листинге 11.22, но с использованием `Window`, потребуется код листинга 11.23.

Листинг 11.23. Сглаживание с помощью `Window`

```
IObservable<Point> smoothed =
    from points in dragPositions.Window(5, 2)
    from totals in points.Aggregate(
        new { X = 0.0, Y = 0.0, Count = 0 },
        (acc, point) => new
            { X = acc.X + point.X, Y = acc.Y + point.Y, Count = acc.Count + 1 })
    where totals.Count > 0
    select new Point(totals.X / totals.Count, totals.Y / totals.Count);
```

Его код немного сложнее, потому что из-за необходимости учитывать появление пустых окон я не смог использовать оператор `Average`. (Собственно говоря, это не имеет значения, если у меня есть одна `Polyline`, которая становится все длиннее и длиннее. Но если группирую точки с помощью операции перетаскивания, как в листинге 11.20, каждый отдельный наблюдаемый источник точек завершит работу в конце перетаскивания, заставив меня обрабатывать все пустые окна.) Оператор `Average` выдает ошибку, если вы передадите ему пустую последовательность, поэтому я использовал вместо него оператор `Aggregate`, который позволяет добавить выражение `where` для фильтрации пустых окон вместо получения сбоев. Но это не единственный аспект, который добавляет сложности.

Как я упоминал ранее, все агрегирующие операторы Rx — `Aggregate`, `Min`, `Max` и т. д. — работают иначе, чем в большинстве провайдеров LINQ. LINQ требует, чтобы эти операторы сокращали поток до единственного значения, поэтому они, как правило, возвращают одно значение. Например, если бы я вызвал версию `Aggregate` для LINQ to Objects с аргументами, показанными в листинге 11.23, оператор возвратил бы одно значение анонимного типа,

который я использую для своего аккумулятора. Но в Rx тип возвращаемого значения — `IObservable<T>` (где `T` — это в данном случае тип аккумулятора). Он по-прежнему выдает единственное значение, но представляет это значение в виде наблюдаемого источника. В отличие от LINQ to Objects, который может перечислять свои входные данные для вычисления, скажем, среднего значения, оператор Rx должен ждать, пока источник предоставит свои значения, и поэтому не может произвести агрегирование этих значений, пока источник не сообщит о завершении работы.

Поскольку оператор `Aggregate` возвращает `IObservable<T>`, мне пришлось использовать второе выражение `from`. Благодаря этому источник передается оператору `SelectMany`, который извлекает из него все значения и передает их в итоговый поток. В данном случае имеется только одно значение (на окно), поэтому `SelectMany` по сути разворачивает усредненную точку из одноэлементного потока.

Код в листинге 11.23 немного сложнее, чем в листинге 11.22, и я считаю, что понять его работу значительно сложнее. Хуже того, он даже не дает никакого выигрыша. Оператор `Aggregate` начнет работу, как только станут доступны входные данные, но код не может вычислить конечный результат — среднее значение, — пока не станет доступна каждая точка в окне. Если мне приходится ждать до конца окна, прежде чем обновить пользовательский интерфейс, я с тем же успехом могу использовать `Buffer`. Так что в данном конкретном случае `Window` приводит к большему объему работы без какой-либо выгоды. Однако если бы работа над элементами в окне была менее тривиальной или если бы объемы данных оказались настолько велики, что вам бы не захотелось буферизовать все окно перед началом обработки, дополнительная сложность могла бы окупиться за счет возможности запуска процесса агрегации до того, как все окно ввода станет доступным.

Разграничение окон с помощью наблюдаемых объектов

Операторы `Window` и `Buffer` обладают и другими способами определения момента, когда окна должны начинаться и заканчиваться. Подобно тому как операторы объединения могут указывать время работы с помощью наблюдаемого источника, вы можете предоставить функцию, которая возвращает для каждого окна определяющий время работы наблюдаемый объект. Листинг 11.24 использует эту технику, чтобы разбить ввод с клавиатуры на слова. Переменная `keySource` пришла в него из листинга 11.11. Это наб-

людаемая последовательность, которая выдает один элемент для каждого нажатия клавиши.

Листинг 11.24. Разбиение текста на слова с помощью окон

```
IObserverable<IObserverable<char>> wordWindows = keySource.Window(  
    () => keySource.FirstAsync(char.IsWhiteSpace));  
  
IObserverable<string> words = from wordWindow in wordWindows  
    from chars in wordWindow.ToArray()  
    select new string(chars).Trim();  
  
words.Subscribe(word => Console.WriteLine("Word: " + word));
```

В этом примере оператор `Window` немедленно создаст новое окно, а также вызовет лямбду, которую я предоставил, чтобы узнать, когда окно должно закончиться. Он будет держать его открытым, пока наблюдаемый источник, который возвращает моя лямбда, не выдаст значение или не завершится. Когда это произойдет, `Window` немедленно откроет следующее окно, снова вызвав мою лямбду, чтобы получить другой наблюдаемый объект для определения длины второго окна, и т. д. В данном случае лямбда выдает следующий символ пробела с клавиатуры, поэтому окно завершится на следующем пробеле. Другими словами, пример разбивает входную последовательность на серию окон, где каждое окно содержит ноль или более отличных от пробела символов, за которыми следует один символ пробела.

Наблюдаемая последовательность, которую возвращает оператор `Window`, представляет каждое окно как `IObserverable<char>`. Второе утверждение в листинге 11.24 — это запрос, который преобразует каждое окно в строку. (Он приведет к появлению пустых строк, если ввод содержит несколько смежных символов пробела. Это согласуется с поведением метода `Split` типа `string`, который выполняет активный эквивалент этого разбиения. Если вам это не подходит, всегда можно отфильтровать пробелы с помощью выражения `where`.)

В листинге 11.24 используется `Window`, который делает символы каждого слова доступными, как только пользователь их набирает. Но поскольку мой запрос вызывает `ToArray` окна, в конечном итоге он будет ждать завершения окна, прежде чем что-либо выдавать. Это означает, что `Buffer` в данном случае тоже подойдет, не говоря уже о том, что его использовать проще. Как показано в листинге 11.25, при использовании `Buffer` второе выражение

`from` для получения завершенного окна не понадобится, потому что он и так предоставляет окна только после их завершения.

Листинг 11.25. Разбиение на слова с помощью Buffer

```
IObservable<IList<char>> wordWindows = keySource.Buffer(  
    () => keySource.FirstAsync(char.IsWhiteSpace));  
  
IObservable<string> words = from wordWindow in wordWindows  
    select new string(wordWindow.ToArray()).Trim();
```

Оператор Scan

Оператор сканирования очень похож на стандартный оператор агрегирования, но с одним отличием. Вместо получения одного результата после завершения его источника он создает последовательность, содержащую значения аккумулятора друг за другом. Чтобы проиллюстрировать это, я напишу класс, который будет действовать как очень простая модель для торговли акциями. Этот класс, показанный в листинге 11.26, кроме прочего, определяет статический метод, который для целей тестирования предоставляет случайно сгенерированный поток сделок.

Листинг 11.26. Простая торговля акциями с тестовым потоком

```
public class Trade  
{  
    public string StockName { get; set; }  
    public decimal UnitPrice { get; set; }  
    public int Number { get; set; }  
  
    public static IObservable<Trade> TestStream()  
    {  
        return Observable.Create<Trade>(obs =>  
        {  
            string[] names = { "MSFT", "GOOGL", "AAPL" };  
            var r = new Random(0);  
            for (int i = 0; i < 100; ++i)  
            {  
                var t = new Trade  
                {  
                    StockName = names[r.Next(names.Length)],  
                    UnitPrice = r.Next(1, 100),  
                    Number = r.Next(10, 1000)  
                };  
                obs.OnNext(t);  
            }  
        });  
    }  
}
```

```
        obs.OnNext(t);
    }
    obs.OnCompleted();
    return default(IDisposable);
});
```

В листинге 11.27 показан обычный оператор `Aggregate`, используемый для расчета общего числа торгуемых акций путем сложения свойства `Number` каждой сделки. (Обычно для этого используется оператор `Sum`, но я привожу данный пример для сравнения со `Scan`.)

Листинг 11.27. Суммирование с помощью Aggregate

```
Iobservable<Trade> trades = Trade.TestStream();  
  
Iobservable<long> tradeVolume = trades.Aggregate(  
    0L, (total, trade) => total + trade.Number);  
tradeVolume.Subscribe(Console.WriteLine);
```

Код отображает одно число, потому что наблюдаемый источник, созданный **Aggregate**, предоставляет единственное значение. В листинге 11.28 показан почти такой же код, но с использованием **Scan**.

Листинг 11.28. Итог с помощью Scan

```
IObservable<Trade> trades = Trade.TestStream();  
  
IObservable<long> tradeVolume = trades.Scan(  
    0L, (total, trade) => total + trade.Number);  
tradeVolume.Subscribe(Console.WriteLine);
```

Вместо того чтобы создавать одно выходное значение, код создает один выходной элемент для каждого входа, и он является промежуточной суммой для всех элементов, выданных источником. `Scan` особенно полезен, если требуется похожее на агрегацию поведение, но для бесконечного потока, например источника событий. `Aggregate` тут не поможет, потому что он ничего не выдает, пока его ввод не завершится.

Оператор Amb

Rx определяет оператор с загадочным названием `Amb`. (См. врезку: «Почему `Amb`?».) Он принимает любое количество наблюдаемых последовательностей и ждет, какая из них сделает что-то первой. (Согласно документации,

какой из входов «прореагирует» первым, т. е. вызовет один из трех методов `IObserver<T>`.) Какой бы ввод ни сработал первым, он фактически становится выходом оператора `Amb`, перенаправляя все, что делает выбранный поток, и немедленно отписываясь от других потоков. (Если какому-то из них удастся выдать элементы после первого потока, но до того, как оператор успел отписаться, эти элементы будут проигнорированы.)

ПОЧЕМУ AMB?

Название оператора `Amb` — это сокращение от `ambiguous` (неоднозначный). Это выглядит нарушением собственных правил разработки библиотеки классов компании Microsoft, которые запрещают сокращения, если только сокращенная форма не используется более широко, чем полное имя, и по этой причине будет понятна даже неопытным специалистам. Имя этого оператора хорошо известно — оно впервые появилось в 1963 году в статье Джона Маккарти (изобретателя языка LISP). Тем не менее оно не так уж широко используется, поэтому вряд ли пройдет проверку на понятность среди неопытных разработчиков.

Но следует признать, что полное имя тоже не так уж и понятно. Если вы впервые видите этот оператор, имя `Ambiguous` поможет разобраться в его работе не лучше, чем просто `Amb`. Если же вы знакомы с ним, вы уже знаете, что он называется `Amb`. Таким образом, в использовании аббревиатуры нет особых недостатков, но есть преимущество для тех, кто о нем уже знает.

Другая причина, по которой команда Rx использовала это имя, заключается в желании воздать должное Джону Маккарти, чья работа оказала огромное влияние на вычисления в целом и на проекты LINQ и Rx в частности. (На многие из функций, обсуждаемых в этой главе и в главе 10, напрямую повлияли работы Маккарти.)

Вы можете использовать этот оператор для оптимизации времени отклика системы, отправив запрос нескольким машинам в пуле серверов и используя результат от того, который ответит первым. (Конечно, эта техника небезопасна. Не в последнюю очередь из-за того, что она может настолько увеличить общую нагрузку на вашу систему, что в результате все замедлится, а не ускорится. Однако существует ряд сценариев, в которых продуманное применение этой техники может дать результат.)

DistinctUntilChanged

Последний оператор, который я собираюсь описать в этом разделе, прост, но довольно полезен. Оператор `DistinctUntilChanged` удаляет соседние дубли-

каты. Предположим, у вас есть наблюдаемый источник, который регулярно выдает элементы, но имеет тенденцию выдавать одно и то же значение несколько раз подряд. Возможно, вам нужно выполнять работу, только когда появляется отличное значение. `DistinctUntilChanged` предназначен именно для этого — когда его вход выдает элемент, он будет передаваться только в том случае, если отличается от предыдущего (или если он первый).

Пока я показал все операторы Rx, которые хотел. Однако остальные, о которых я расскажу в разделе «Операции для работы со временем» на с. 658, все чувствительны ко времени. И прежде, чем перейти к ним, я должен рассказать, как Rx управляет временем.

Планировщики

Rx проделывает определенную работу с помощью планировщиков. Планировщик — это объект, который выполняет три задачи. Во-первых, он решает, когда выполнять конкретную часть работы. Например, когда наблюдатель подписывается на холодный источник, должны ли элементы источника доставляться подписчику немедленно или же эту работу следует отложить? Вторая задача — запуск работы в определенном контексте. Например, планировщик может решить выполнять всю работу в определенном потоке. Третья задача — учет времени. Некоторые операции Rx зависят от времени; чтобы обеспечить предсказуемое поведение и дать возможность тестирования, планировщики предоставляют виртуализированную модель учета времени, поэтому код Rx перестает зависеть от текущего времени суток, сообщаемого классом .NET `DateTimeOffset`.

Первые две роли планировщика иногда взаимозависимы. Например, Rx предоставляет несколько планировщиков для использования в приложениях пользовательского интерфейса. Существует `CoreDispatcherScheduler` для приложений Windows Store, `DispatcherScheduler` — для приложений WPF, `Control Scheduler` — для программ Windows Forms и более обобщенный `SynchronizationContextScheduler`, который будет работать со всеми библиотеками разработки интерфейса .NET, но даст несколько меньший контроль над деталями, нежели специализированные варианты³. Все они имеют общую

³ Это имя (которое сейчас выглядит устаревшим) происходит из того факта, что некоторое время приложения Windows Store оставались единственной поддерживаемой средой для использования .NET Core. Тем не менее `CoreDispatcherScheduler` недоступен ни в какой другой платформе на основе .NET Core.

характеристику: они гарантируют, что работа выполняется в подходящем контексте с доступом к объектам пользовательского интерфейса, что обычно означает работу в конкретном потоке. Если код, который планирует работу, выполняется в каком-то другом потоке, у планировщика может не быть иного выбора, кроме как отложить работу, потому что он не сможет запустить ее до готовности пользовательского интерфейса. Это может означать ожидание, пока определенный поток завершит свою текущую задачу. В данном случае выполнение работы в правильном контексте неизбежно влияет на то, когда именно выполняется работа.

Однако это не всегда так. Rx предоставляет два планировщика, которые используют текущий поток. Один из них, `ImmediateScheduler`, чрезвычайно прост: он запускает работу в тот момент, когда она запланирована. Когда вы дадите этому планировщику некоторую работу, он не завершится, пока не будет завершена работа. Другой, `CurrentThreadScheduler`, поддерживает рабочую очередь, что дает ему некоторую гибкость в планировании. Например, если какая-то работа должна запуститься в середине выполнения какой-либо другой части работы, он может дать выполняемой задаче завершиться до запуска следующей. Если нет выполняющихся задач и задач в очереди, `CurrentThreadScheduler` немедленно запускает работу, как и `ImmediateScheduler`. Когда вызванная задача завершается, `CurrentThreadScheduler` проверяет очередь и вызывает следующий элемент, если она не пустая. Иными словами, он пытается завершить всю работу как можно быстрее, но, в отличие от `ImmediateScheduler`, не начнет обрабатывать новую задачу до того, как завершится предыдущая.

Задание планировщиков

Операции Rx часто не проходят через планировщики. Многие наблюдаемые источники вызывают методы своих подписчиков напрямую. Как правило, исключением являются источники, которые способны генерировать большое количество элементов в быстрой последовательности. Например, методы `Range` и `Repeat` для создания последовательностей используют планировщик для управления скоростью, с которой они предоставляют элементы новым подписчикам. Вы можете передать им явный планировщик или позволить выбор по умолчанию. Вы также можете явно задействовать планировщик, даже если используете источники, которые не принимают его в качестве аргумента.

ObserveOn

Распространенным способом задания планировщика является использование одного из методов расширения `ObserveOn`, определяемого различными статическими классами в пространстве имен `System.Reactive.Linq`. Он полезен, когда необходимо обрабатывать события в определенном контексте (например, в потоке пользовательского интерфейса), даже если они могут приходить откуда-то еще⁴.

Можно вызвать `ObserveOn` для любого `IEnumerable<T>`, передав `IScheduler`, и он вернет другой `IEnumerable<T>`. Если вы подпишитесь на возвращаемый наблюдаемый объект, все методы `OnNext`, `OnCompleted` и `OnError` вашего наблюдателя будут вызываться через указанный вами планировщик. Листинг 11.29 использует эту технику для гарантии безопасного обновления пользовательского интерфейса в обратном вызове обработчика элемента.

Листинг 11.29. ObserveOn

```
IEnumerable<Trade> trades = GetTradeStream();
IEnumerable<Trade> tradesInUiContext =
    trades.ObserveOn(DispatcherScheduler.Current);
tradesInUiContext.Subscribe(t =>
{
    tradeInfoTextBox.AppendText(
        $"{t.StockName}: {t.Number} at {t.UnitPrice}\r\n");
});
```

В этом примере я использовал статическое свойство `Current` класса `DispatcherScheduler`, которое возвращает планировщик, выполняющий работу через `Dispatcher` текущего потока. (`Dispatcher` — это класс, который управляет циклом сообщений пользовательского интерфейса в приложениях WPF.) Существует альтернативная перегрузка `ObserveOn`, которую можно в данном случае использовать. Класс `DispatcherObservable` определяет некоторые методы расширения, обеспечивающие специфичные для WPF перегрузки, что позволяет мне вызывать `ObserveOn`, передавая только объект `Dispatcher`. Я мог бы использовать это в коде предыдущего примера для элемента пользовательского интерфейса, применив код, показанный в листинге 11.30.

⁴ Перегрузки распределены по нескольким классам, потому что некоторые из этих методов расширения зависят от определенных технологий. WPF содержит перегрузки `ObserveOn`, которые, например, работают непосредственно с его классом `Dispatcher` вместо `IScheduler`.

Листинг 11.30. Специфичная для WPF перегрузка ObserveOn

```
IObservable<Trade> tradesInUiContext = trades.ObserveOn(this.Dispatcher);
```

Преимущество этой перегрузки заключается в том, что при ее использовании, в момент вызова `ObserveOn`, мне не нужно находиться в потоке пользовательского интерфейса. Свойство `Current`, используемое в листинге 11.29, работает только в том случае, если вы находитесь в потоке нужного вам диспетчера. Если мы уже и так в этом потоке, т. е. еще более простой способ. Можно использовать метод расширения `ObserveOnDispatcher`, который получает `DispatcherScheduler` для диспетчера текущего потока, как показано в листинге 11.31.

Листинг 11.31. Наблюдение за текущим диспетчером

```
IObservable<Trade> tradesInUiContext = trades.ObserveOnDispatcher();
```

SubscribeOn

Большинство различных методов расширения `ObserveOn` имеют соответствующие методы `SubscribeOn`. (Существует также `SubscribeOnDispatcher`, аналог `ObserveOnDispatcher`.) Вместо того чтобы заниматься каждым вызовом методов наблюдателя, выполняемым через планировщик, `SubscribeOn` выполняет вызов через планирощик метода `Subscribe` исходного наблюдаемого объекта. И если вы отмените подписку, вызвав `Dispose`, это также будет осуществлено через планировщик. Это важно для холодных источников, потому что многие из них выполняют значительный объем работы в своем методе `Subscribe`, а некоторые вообще немедленно выдают все свои элементы.



В целом, нет никакой гарантии соответствия между контекстом, в котором вы подписываетесь на источник, и контекстом, в котором выдающиеся им элементы будут доставлены подписчику. Некоторые источники будут отправлять уведомления из контекста подписки, но многие — нет. Если вам требуется получать уведомления в определенном контексте, то в случае, если источник не предоставляет способа указать планировщик, используйте `ObserveOn`.

Явная передача планировщиков

Некоторые операции принимают планировщик в качестве аргумента. Это часто встречается в операциях, которые способны генерировать много элементов. С целью управления контекстом, из которого генерируются числа,

метод `Observable.Range` дополнительно принимает планировщик в качестве последнего аргумента. Это также относится к API для адаптации других источников, таких как `IEnumerable<T>`, к наблюдаемым источникам, что описано в разделе «Адаптация» на с. 650.

Другой сценарий, в котором вы, как правило, предоставляете планировщик, — это использование наблюдаемого объекта, который объединяет входные данные. Ранее вы видели, как оператор `Merge` объединяет вывод нескольких последовательностей. Вполне можно допустить существование планировщика, который предлагает оператору подписаться на источники из определенного контекста.

Наконец, от планировщика зависят все синхронизированные операции. Я покажу некоторые из них в разделе «Операции для работы со временем» на с. 658.

Встроенные планировщики

Я уже описал четыре планировщика, ориентированных на пользовательский интерфейс, `DispatcherScheduler` (для WPF), `CoreDispatcherScheduler` (для приложений Windows Store), `ControlScheduler` (для Windows Forms) и `SynchronizationContextScheduler`, а также два планировщика для выполнения задач в текущем потоке, `CurrentThreadScheduler` и `ImmediateScheduler`. Но существует и ряд других, о которых стоит знать.

`EventLoopScheduler` запускает все задачи в определенном потоке. Он может создать новый поток за вас, или же вы можете предоставить ему метод обратного вызова, к которому он обратится, когда потребуется, чтобы поток создали вы. Вы можете использовать это в приложении с пользовательским интерфейсом для обработки входящих данных. Это позволит вам убрать ваши задачи из потока пользовательского интерфейса, чтобы сохранить отзывчивость приложения, и при этом даст гарантию, что вся обработка будет происходить в одном потоке, что потенциально снимет вопросы параллелизма.

`NewThreadScheduler` создает новый поток для каждой задачи верхнего уровня, которую он обрабатывает. (Если эта задача порождает дополнительные задачи, они будут выполняться в том же потоке, а не в новых.) Это сработает только в том случае, если вам требуется выполнять большой объем работы для каждого элемента, поскольку потоки в Windows имеют относительно высокую стоимость запуска и остановки. Если вам нужна параллельная обработка задач, хорошим решением будет использование пула потоков.

`TaskPoolScheduler` использует пул потоков `Task Parallel Library` (`TPL`). `TPL`, описанный в главе 16, предоставляет действующий пул потоков, который может повторно использовать один поток для нескольких задач, снижая затраты на создание потока.

`ThreadPoolScheduler` для запуска задачи использует пул потоков `CLR`. В принципе, он похож на пул потоков `TPL`, просто это чуть более старая технология. (`TPL` появился в .NET 4.0, а пул потоков `CLR` существует с версии 1.0.) В некоторых сценариях он немного менее эффективен. Этот планировщик добавлен в Rx, потому что ранние версии Rx поддерживали старые версии .NET, которые не имели `TPL`. Так что он сохраняется для обратной совместимости.

`HistoricalScheduler` полезен, когда нужно протестировать чувствительный ко времени код без необходимости выполнять тесты в режиме реального времени. Все планировщики будут предоставлять возможность учета времени, но `HistoricalScheduler` позволяет вам задать точную скорость, с которой для вашего планировщика будет идти время. Итак, если вам нужно проверить, что произойдет, если ожидание длится 30 секунд, вам достаточно просто указать `HistoricalScheduler`, что прошло 30 секунд, без необходимости ждать на самом деле.

Субъекты

В Rx определены различные субъекты, классы, которые реализуют как `IObserver<T>`, так и `Observable<T>`. Это может быть кстати, если нужно, чтобы Rx обеспечил надежную реализацию любого из этих интерфейсов, но обычные методы `Observable.Create` или `Subscribe` не подходят. Например, вам может понадобиться предоставить наблюдаемый источник, и в вашем коде есть несколько разных мест, из которых вы хотите предоставить значения для этого источника. Это едва ли вписывается в модель обратного вызова подписки метода `Create`, и гораздо проще решить эту проблему с помощью субъекта. Некоторые типы субъектов обеспечивают дополнительное поведение, но я начну с самого простого, `Subject<T>`.

`Subject<T>`

Реализация `IObserver<T>` класса `Subject<T>` просто передает вызовы всем наблюдателям, которые подписались с помощью интерфейса `IObservable<T>`. Таким образом, если вы подписываете один или несколько наблюдаемых

объектов на `Subject<T>`, а затем вызываете `OnNext`, субъект вызовет `OnNext` каждого из своих подписчиков. Это же справедливо и для других методов, `OnCompleted` и `OnError`. Такая многоадресная передача очень похожа на ту, которую предоставляет оператор `Publish`, который я использовал в листинге 11.11, так что это дает мне еще один способ удалить весь код для отслеживания подписчиков из моего источника `KeyWatcher`⁵. В результате получается код, показанный в листинге 11.32. Он намного проще, чем оригинал в листинге 11.7, хотя и не так прост, как версия на основе делегатов в листинге 11.11.

Листинг 11.32. Реализация `IObservable<T>` с помощью `Subject<T>`

```
public class KeyWatcher : IObserverable<char>
{
    private readonly Subject<char> _subject = new Subject<char>();

    public IDisposable Subscribe(IObserver<char> observer)
    {
        return _subject.Subscribe(observer);
    }

    public void Run()
    {
        while (true)
        {
            _subject.OnNext(Console.ReadKey(true).KeyChar);
        }
    }
}
```

В своем методе `Subscribe` он полагается на `Subject<char>`, поэтому все объекты, которые пытаются подписаться на `KeyWatcher`, в конечном итоге подпишутся на этот субъект. Мой цикл может просто вызвать метод субъекта `OnNext`, а он уже позаботится о том, чтобы передать этот вызов всем подписчикам.

На самом деле я могу упростить все еще больше, выделив наблюдаемый объект как отдельное свойство, вместо того чтобы сделать наблюдаемым весь мой тип, как это показано в листинге 11.33. Это не только сделает код немного проще, но и будет означать, что мой `KeyWatcher` теперь способен, если нужно, предоставлять несколько источников.

⁵ Фактически в текущей версии Rx внутри `Publish` используется `Subject<T>`.

Листинг 11.33. Предоставление `IObservable<T>` в виде свойства

```
public class KeyWatcher
{
    private readonly Subject<char> _subject = new Subject<char>();

    public Iobservable<char> Keys => _subject;

    public void Run()
    {
        while (true)
        {
            _subject.OnNext(Console.ReadKey(true).KeyChar);
        }
    }
}
```

Этот подход по-прежнему нельзя сравнивать по простоте с сочетанием `Observable.Create` и оператора `Publish`, которое я использовал в листинге 11.11, но он дает два преимущества. Во-первых, теперь легче понять, когда запускается цикл, генерирующий уведомления о нажатии клавиши. В листинге 11.11 этим управляем я, но для любого, кто не так хорошо знаком с работой `Publish`, было не очевидно, каким образом это достигалось. По-моему, листинг 11.33 уже не такой запутанный. Во-вторых, если бы я захотел, я мог бы использовать этот субъект где угодно внутри моего класса `KeyWatcher`. Тогда как в листинге 11.11 единственное место, из которого я мог без проблем предоставить элемент, находилось внутри функции обратного вызова, вызываемой `Observable.Create`. Так получилось, что в этом примере мне не нужна подобная гибкость, но в сценариях, где она нужна, `Subject<T>`, вероятно, будет более правильным выбором, нежели подход с обратным вызовом.

BehaviorSubject<T>

`BehaviorSubject<T>` выглядит близнецом `Subject<T>`, за исключением одного: когда любой наблюдатель впервые на него подписывается, до завершения субъекта с помощью метода `OnComplete` он гарантированно и сразу получает значение. (Если вы уже завершили субъект, он попросту немедленно вызывает `OnComplete` для всех последующих подписчиков.) Он запоминает последний переданный элемент и передает его новым подписчикам. При создании `BehaviorSubject<T>` вы должны дать ему начальное значение, которое он будет предоставлять новым подписчикам до первого вызова `OnNext`.

Можно думать об этом субъекте как о переменной по версии Rx. Это не-что, что содержит значение, которое можно получить в любое время, и это значение со временем может поменяться. Но с реактивным подходом для получения значения вы на него подписываетесь и ваш наблюдатель будет уведомляться о любых дальнейших изменениях, вплоть до отмены подписки.

Этот субъект сочетает в себе характеристики горячих и холодных объектов. Он будет мгновенно предоставлять значение любому подписчику, что делает его похожим на холодный источник, но после этого будет транслировать новые значения всем подписчикам, что выглядит уже как горячий источник. Есть еще один субъект с похожим поведением, но он немного уклоняется в сторону холодного.

ReplaySubject<T>

`ReplaySubject<T>` может записывать каждое значение, полученное из любого источника, на который вы подписаны. (Или, если вы вызываете его методы напрямую, он запоминает каждое значение, предоставленное вами через `OnNext`.) Каждый новый подписчик на этот субъект будет получать каждый элемент, который `ReplaySubject<T>` получил до момента подписки. Это больше похоже на обычный холодный субъект — вместо того чтобы просто получить самое последнее значение, как если бы это был `BehaviorSubject<T>`, вы получаете полный набор элементов. Однако как только `ReplaySubject<T>` предоставил конкретному подписчику все элементы, которые он записал, он переходит к более похожему на горячее поведению в отношении этого подписчика, продолжая предоставлять новые входящие элементы.

В конечном счете каждый подписчик на `ReplaySubject<T>` по умолчанию получит каждый элемент, который `ReplaySubject<T>` получил от своего источника, независимо от того, когда именно он подписался на этот субъект.

В конфигурации по умолчанию `ReplaySubject<T>` будет потреблять все больше памяти, пока остается подписанным на источник. Нет никакого способа сообщить ему, что у него больше не будет новых подписчиков и что теперь можно избавиться от старых элементов, которые он уже разоспал всем своим существующим подписчикам. Поэтому его нельзя навсегда оставлять подписанным на бесконечный источник. Тем не менее вы можете ограничить размер буфера `ReplaySubject<T>`. Субъект предлагает различные перегрузки конструктора, некоторые из которых позволяют вам указать либо верхний

предел количества элементов для воспроизведения, либо верхний предел времени, в течение которого он будет хранить эти элементы. Очевидно, что если вы это сделаете, новые подписчики больше не смогут полагаться на получение всех ранее полученных элементов.

AsyncSubject<T>

`AsyncSubject<T>` запоминает только одно значение из своего источника, но, в отличие от `BehaviorSubject<T>`, который запоминает самое недавнее, `AsyncSubject<T>` ожидает завершения своего источника. После этого он выдает последний элемент в качестве вывода. Если источник завершает работу без предоставления каких-либо значений, `AsyncSubject<T>` поступит так же со своими подписчиками.

Если вы подписались на `AsyncSubject<T>` до завершения его источника, `AsyncSubject<T>` не будет делать с вашим наблюдателем ничего до завершения работы источника. Но, как только работа источника оказывается завершена, `AsyncSubject<T>` действует как холодный источник, который предоставляет единственное значение. Если же источник закончил работу без выдачи значения, этот субъект немедленно завершит работу всех новых подписчиков.

Адаптация

Несмотря на всю свою мощь и привлекательность, Rx не сильно полезна в изоляции. Если вы работаете с асинхронными уведомлениями, вполне возможно, что их предоставляет API, который не поддерживает Rx. Хотя `IObservable<T>` и `IObserver<T>` существуют уже давно (начиная с версии .NET 4.0, выпущенной в 2010 году), не все API, которые могли бы поддерживать эти интерфейсы, их на самом деле поддерживают. Кроме того, поскольку фундаментальная абстракция Rx — это последовательность элементов, есть хороший шанс, что в какой-то момент вам может понадобиться выполнить преобразование пассивного `IEnumerable<T>` из Rx в активные эквиваленты, `IEnumerable<T>` и `IAsyncEnumerable<T>`. Rx предоставляет способы для преобразования всех этих видов источников в `IObservable<T>`, а в некоторых случаях он способен осуществить преобразование в обратном направлении.

IEnumerable<T> и IAsyncEnumerable<T>

Любой `IEnumerable<T>` может быть легко перенесен в среду Rx с помощью методов расширения `ToObservable`. Они определяются статическим классом `Observable` в пространстве имен `System.Reactive.Linq`. Листинг 11.34 показывает простейшую форму, которая не принимает аргументов.

Листинг 11.34. Преобразование `IEnumerable<T>` в `IObservable<T>`

```
public static void ShowAll(IEnumerable<string> source)
{
    IObservable<string> observableSource = source.ToObservable();
    observableSource.Subscribe(Console.WriteLine);
}
```

Сам по себе метод `ToObservable` ничего не делает с входными данными — он просто возвращает обертку, которая реализует `IObservable<T>`. Эта обертка является холодным источником, и каждый раз, когда вы подписываете на нее наблюдателя, она выполняет перебор входных данных, передавая каждый элемент в метод `OnNext` наблюдателя и вызывая `OnCompleted` в самом конце. Если источник выдает исключение, этот адаптер вызовет `OnError`. Листинг 11.35 показывает, как мог бы работать `ToObservable`, если бы ему не требовалось использовать планировщик.

Листинг 11.35. Как бы выглядел `ToObservable` без поддержки планировщика

```
public static IObservable<T> MyToObservable<T>(this IEnumerable<T> input)
{
    return Observable.Create<T>(IObserver<T> observer) =>
    {
        bool inObserver = false;
        try
        {
            foreach (T item in input)
            {
                inObserver = true;
                observer.OnNext(item);
                inObserver = false;
            }
            inObserver = true;
            observer.OnCompleted();
        }
        catch (Exception x)
        {
            if (inObserver)
```

```

    {
        throw;
    }
    observer.OnError(x);
}
return () => { };
});
}

```

На самом деле он работает не совсем так — в этом коде нельзя использовать планировщик. (Полноценную реализацию было бы намного сложнее читать, тогда как пример призван показать основную идею, лежащую в основе `ToObservable`.) Реальный метод использует планировщик для управления процессом перебора, позволяя при необходимости выполнять подписку асинхронно. Он также поддерживает остановку работы, если подписка наблюдателя отменена досрочно. У него имеется перегрузка, принимающая один аргумент типа `IScheduler`, который позволяет указать конкретный планировщик; если вы его не предоставите, будет использован `CurrentThreadScheduler`.

Что касается преобразования в обратном направлении, т. е. когда у вас `Observable<T>`, но вы хотели работать с ним как с `IEnumerable<T>`, вы можете вызвать методы расширения `ToEnumerable`, также предоставляемые классом `Observable`. Листинг 11.36 создает обертку для `IEnumerable<string>`, превращая его в `IEnumerable<string>`, в результате чего может перебирать элементы в источнике, используя обычный цикл `foreach`.

Листинг 11.36. Использование `IEnumerable<T>` как `IEnumerable<T>`

```

public static void ShowAll(IEnumerable<string> source)
{
    foreach (string s in source.ToEnumerable())
    {
        Console.WriteLine(s);
    }
}

```

Обертка подписывается на источник от вашего имени. Если источник выдает элементы быстрее, чем вы способны их перебирать, оболочка сохранит элементы в очереди, чтобы вы могли получить их в нужное время. Если же источник не предоставляет элементы достаточно быстро, обертка будет просто ожидать, пока элементы не станут доступными.

В .NET Core 3.0 и .NET Standard 2.1 добавлен `IAsyncEnumerable<T>`, интерфейс, который поддерживает тот же функционал, что и `IEnumerable<T>`, но так, чтобы обеспечить эффективную асинхронную работу с использованием методов, описанных в главе 17. Для этого в Rx содержится метод расширения `ToObservable`, а также метод расширения метода `ToAsyncEnumerable` для `IEnumerable<T>`. Они оба происходят из класса `AsyncEnumerable`, и для его использования вам потребуется ссылка на отдельный пакет NuGet под названием `System.Linq.Async`.

События .NET

Rx может обернуть событие .NET в виде `IObservable<T>`, используя статический метод `FromEventPattern` класса `Observable`. Ранее в листинге 11.17 я использовал `FileSystemWatcher`, класс из пространства имён `System.IO`, который вызывает различные события, когда файлы добавляются, удаляются, переименовываются или иным образом изменяются в определенной папке. Листинг 11.37 воспроизводит первую часть примера, по которому я пробежался в прошлый раз. Этот код использует статический метод `Observable.FromEventPattern` для создания наблюдаемого источника, представляющего событие `Created` наблюдателя. (Если требуется обработать статическое событие, вы можете передать объект `Type` в качестве первого аргумента. Класс `Type` описывается в главе 13.)

Листинг 11.37. Оборачивание события в `IObservable<T>`

```
string path = Environment.GetFolderPath(
    Environment.SpecialFolder.MyPictures);
var watcher = new FileSystemWatcher(path);
watcher.EnableRaisingEvents = true;

IObservable<EventPattern<FileSystemEventArgs>> changes =
    Observable.FromEventPattern<FileSystemEventArgs>(
        watcher, nameof(watcher.Created));
changes.Subscribe(evt => Console.WriteLine(evt.EventArgs.FullPath));
```

На первый взгляд код выглядит значительно сложнее, чем обычная подписка на событие, показанная в главе 9, причем без очевидного преимущества. И в этом конкретном примере можно было постараться и получше. Однако одно очевидное преимущество использования Rx состоит в том, что если вы пишете приложение с пользовательским интерфейсом, вы можете использовать `ObserveOn` с подходящим планировщиком. Это гарантирует, что

ваш обработчик всегда будет вызываться в нужном потоке независимо от того, какой поток вызвал событие. Еще одно преимущество — и обычная причина для применения этого подхода — состоит в том, что для обработки событий вы можете использовать любой из операторов запросов Rx. (Вот почему это делается в оригинальном листинге 11.17.)

Тип элемента наблюдаемого источника, который генерирует листинг 11.37, — `EventPattern<FileSystemEventArgs>`.

`EventPattern<T>` является обобщенным типом, определяемым Rx специально для представления события, где тип делегата события соответствует стандартному шаблону, описанному в главе 9 (т. е. принимает два аргумента, первый из которых является типом `object`, представляющим объект, который вызвал событие, а второй — типом, производным от `EventArgs` и содержащим информацию о событии). `EventPattern<T>` имеет два свойства, `Sender` и `EventArgs`, которые соответствуют двум аргументам, получаемым обработчиком события. По сути, этот объект представляет собой то, что в обычных обстоятельствах было вызовом метода обработчика событий.

Интересная особенность примера 11.37 состоит в том, что вторым аргументом `FromEventPattern` выступает строка, содержащая имя события. Rx разрешает ее в реальное событие уже во время выполнения. Это далеко не идеальный подход по нескольким причинам. Во-первых, это означает, что если вы введете имя неправильно, компилятор этого не заметит (хотя использование оператора `nameof` решает эту проблему). Во-вторых, это означает, что компилятор не сможет помочь вам с типами — если вы обрабатываете событие .NET непосредственно с помощью лямбды, компилятор может получить типы аргументов из определения события. Здесь же, из-за того что мы передаем имя события в виде строки, компилятор не знает, какое именно событие я использую (и использую ли вообще). Именно поэтому мне пришлось явно указать для метода аргумент обобщенного типа. И снова, если я ошибусь, компилятор об этом не узнает — проверка произойдет только во время выполнения.

Этот подход на основе строк имеет место из-за недостатка, присущего событиям: нельзя передать событие в качестве аргумента. На самом деле события — это крайне ограниченные члены. Вы не можете ничего сделать с событием вне определяющего его класса, кроме как добавить или удалить обработчики. Это один из способов, которым Rx улучшает события, — в среде Rx источники событий и подписчики представлены в виде объектов (реа-

лизующих `IObservable<T>` и `IObserver<T>` соответственно), что упрощает передачу их в методы в качестве аргументов. Но это никак нам не поможет, если мы имеем дело с событием, которого еще нет в Rx.

В действительности Rx содержит перегрузку, которая не требует использования строки — вы можете передать делегаты, добавляющие и удаляющие обработчики за Rx, как показано в листинге 11.38.

Листинг 11.38. Оборачивание событий на основе делегатов

```
IObservable<EventPattern<FileSystemEventArgs>> changes =
    Observable.FromEventPattern<FileSystemEventHandler,
    FileSystemEventArgs>(
        h => watcher.Created += h, h => watcher.Created -= h);
```

Она выглядит более многословно, поскольку ей требуется аргумент обобщенного типа, указывающий тип делегата обработчика, а также тип аргумента события. Строковая версия обнаруживает свой тип обработчика во время выполнения, но, поскольку обычная причина для использования подхода из листинга 11.38 состоит в том, чтобы воспользоваться проверкой типов во время компиляции, компилятору нужно знать, какие типы вы используете. Лямбды в этом примере не предоставляют компилятору достаточной информации для автоматического выводения всех типов аргументов.

Наряду с оборачиванием события в наблюдаемый источник можно пойти и в другом направлении. Rx определяет оператор для `IObservable<EventPattern<T>>` под названием `ToEventPattern<T>`. (Обратите внимание, что он будет недоступен для любого старого наблюдаемого источника — это должна быть наблюдаемая последовательность `EventPattern<T>`.) При вызове он возвращает объект, который реализует `IEventPatternSource<T>`. Этот объект определяет единственное событие с именем `OnNext`, типа `EventHandler<T>`, которое позволяет обычным для .NET способом подключить к наблюдаемому источнику обработчик событий.

Обобщенная платформа Windows (UWP, предоставляющая приложениям Windows Store общий API, используемый приложениями .NET и C++) имеет собственный вариант шаблона событий, основанный на типе `TypedEventHandler`. В пространстве имен `System.Reactive.Linq` определен класс `WindowsObservable` с методами преобразования между ними и Rx. (Все это доступно только тогда, когда вы пишете для UWP, — NuGet-пакеты Rx содержат отдельные версии библиотек DLL для различных целевых платформ, из-за чего становятся возможными специфичные для платформ функции,

подобные этим.) Он определяет методы `FromEventPattern` и `ToEventPattern`, которые предоставляют те же возможности, что и описанные выше версии, только для событий UWP вместо обычных событий .NET.

Асинхронные API

.NET поддерживает различные асинхронные модели, которые я подробно опишу в главах 16 и 17. Первой .NET появилась модель асинхронного программирования (APM, Asynchronous Programming Model). Она напрямую не поддерживается новыми функциями асинхронного языка C#, поэтому большинство API .NET в настоящее время используют TPL, а для более старых API TPL содержит адаптеры, которые могут предоставлять обертку на основе задач для API на основе APM. Rx может представить любую задачу TPL в качестве наблюдаемого источника.

Базовой для всех асинхронных моделей .NET является та, в которой вы начинаете некоторую задачу, которая в конечном итоге будет завершена, в ряде случаев выдав результат. Поэтому может показаться странным перевод ее в Rx, где фундаментальная абстракция — это последовательность элементов, но никак не один результат. Разницу между Rx и TPL можно понять так: `IEnumerable<T>` является аналогом `IEnumerable<T>`, тогда как `Task<T>` аналогично свойству типа `T`. Принимая во внимание `IEnumerable<T>` и свойства, вызывающая сторона решает, когда извлекать информацию из источника с помощью `IEnumerable<T>` и `Task<T>`, а источник предоставляет информацию по мере готовности. Выбор того, какая сторона решает, когда предоставлять информацию, совершенно не касается вопроса о форме информации (является ли она одним элементом или целой последовательностью). Таким образом, преобразование между единичными асинхронными API и `IEnumerable<T>` выглядит несколько рассогласованным. Но в неасинхронном мире мы вполне можем выйти за эти границы — как вы видели в главе 10, LINQ определяет различные стандартные операторы, которые выдают один элемент из последовательности, например `First` или `Last`. Rx поддерживает эти операторы, но дополнительно поддерживает и движение в другом направлении: перенос единичных асинхронных источников в мир потоков. В итоге мы получаем источник `IEnumerable<T>`, который выдает только один элемент (или сообщает об ошибке в случае сбоя). Аналог в неасинхронном мире будет принимать одно значение и заключать его в массив, чтобы его можно было передать в API, который требует `IEnumerable<T>`.

Листинг 11.39 использует эту возможность для создания `IObservable<string>`, который либо выдаст одно значение, содержащее текст, загруженный из конкретного URL, либо сообщит об ошибке в случае сбоя.

Листинг 11.39. Оборачивание `Task<T>` в `IObservable<T>`

```
public static IObserverable<string> GetWebPageAsObserverable(
    Uri pageUrl, IHttpClientFactory cf)
{
    HttpClient web = cf.CreateClient();
    Task<string> getPageTask = web.GetStringAsync(pageUrl);
    return getPageTask.ToObserverable();
}
```

Метод `ToObserverable`, используемый в этом примере, является методом расширения, определенным Rx для `Task`. Чтобы он был доступен, вам потребуется пространство имен `System.Reactive.Threading.Tasks`, которое должно находиться в области видимости.

Одна потенциально неприятная особенность листинга 11.39 состоит в том, что он попытается загрузить строку только один раз, вне зависимости от того, сколько наблюдателей подписалось на источник. В зависимости от требований это может быть даже хорошо, но в некоторых случаях разумнее каждый раз пытаться загрузить новую копию. Если вам нужно именно это, лучшим подходом будет использование метода `Observable.FromAsync`, потому что ему вы передаете лямбду, которую он вызывает каждый раз, когда подписывается новый наблюдатель. Ваша лямбда возвращает задачу, которая затем будет обернута в наблюдаемый источник. Листинг 11.40 использует этот подход для запуска новой загрузки для каждого подписчика.

Листинг 11.40. Создание новой задачи для каждого подписчика

```
public static IObserverable<string> GetWebPageAsObserverable(
    Uri pageUrl, IHttpClientFactory cf)
{
    return Observable.FromAsync(() =>
    {
        HttpClient web = cf.CreateClient();
        return web.GetStringAsync(pageUrl);
    });
}
```

Это может работать неоптимально, если у вас много подписчиков. С другой стороны, это работает гораздо более эффективно, когда подписок нет вообще.

Листинг 11.39 немедленно запускает асинхронную работу, не дожидаясь подписчиков. Это может быть и хорошо — если поток точно будет иметь подписчиков, то запуск медленной работы без ожидания первого подписчика уменьшит общую задержку. Однако если вы пишете класс в библиотеке, которая представляет несколько наблюдаемых источников, не все из которых могут использоваться, то откладывание работы до первой подписки может оказаться более выигрышным.

Среда выполнения Windows определяет ряд собственных асинхронных моделей через интерфейсы `IAsyncOperation` и `IAsyncOperationWithProgress`. Пространство имен `System.Reactive.Windows.Foundation` определяет методы расширения для преобразования между ними и Rx. В нем определены методы расширения `ToObservable` для этих типов, а также методы расширения `ToAsyncOperation` и `ToAsyncOperationWithProgress` для `IEnumerable<T>`.

Операции для работы со временем

Поскольку Rx может работать с потоками информации в реальном времени, вам может потребоваться обрабатывать элементы с учетом времени. Например, важным фактором может оказаться скорость поступления элементов или вам может понадобиться группировать элементы по времени их поступления. В этом последнем разделе я опишу некоторые операторы с привязкой ко времени, которые предлагает Rx.

Interval

Метод `Observable.Interval` возвращает последовательность, которая регулярно создает значения в интервале, заданном аргументом типа `TimeSpan`. Листинг 11.41 создает и подписывается на источник, который будет выдавать одно значение каждую секунду.

Листинг 11.41. Регулярная выдача элементов с использованием Interval

```
IEnumerable<long> src = Observable.Interval(TimeSpan.FromSeconds(1));
src.Subscribe(i => Console.WriteLine($"Event {i} at {DateTime.Now:T}"));
```

Элементы, выдаваемые оператором `Interval`, относятся к типу `long`. Он выдает значения ноль, один, два и т. д.

`Interval` обрабатывает каждого подписчика независимо (т. е. это холодный источник). Чтобы продемонстрировать это, добавьте код листинга 11.42 к листингу 11.41, чтобы немного погодя создать вторую подписку.

Листинг 11.42. Два подписчика на один источник `Interval`

```
Thread.Sleep(2500);
src.Subscribe(i => Console.WriteLine(
    $"Event {i} at {DateTime.Now:T} (2nd subscriber)");
```

Второй подписчик подписывается через две с половиной секунды после первого, поэтому в результате мы получим следующий вывод:

```
Event 0 at 09:46:58
Event 1 at 09:46:59
Event 2 at 09:47:00
Event 0 at 09:47:00 (0nd subscriber)
Event 3 at 09:47:01
Event 1 at 09:47:01 (1nd subscriber)
Event 4 at 09:47:02
Event 2 at 09:47:02 (2nd subscriber)
Event 5 at 09:47:03
Event 3 at 09:47:03 (3nd subscriber)
```

Как видите, значения второго подписчика начинаются с нуля из-за того, что он получает свою собственную последовательность. Если вам нужно, чтобы один набор этих синхронизированных элементов передавался нескольким подписчикам, вы можете использовать оператор `Publish`, описанный ранее.

Источник `Interval` можно использовать в сочетании с объединением групп как способ разбить элементы на блоки в зависимости от того, когда они поступают. (Это не единственный способ — существуют перегрузки `Buffer` и `Window`, которые могут делать то же самое.) Листинг 11.43 объединяет таймер с наблюдаемой последовательностью, представляющей слова, которые вводит пользователь. (Эта вторая последовательность находится в переменной `words`, которая взята из листинга 11.25.)

Сгруппировав слова на основе событий из источника `Interval`, этот запрос продолжает работу и подсчитывает количество элементов в каждой группе. Поскольку группы распределены по времени равномерно, это можно использовать для расчета приблизительной скорости, с которой пользователь печатает слова. Я формирую группу каждые 6 секунд, чтобы можно было умножить количество слов в группе на 10 и вычислить количество слов в минуту.

Листинг 11.43. Подсчет слов в минуту

```
IObserveable<long> ticks = Observable.Interval(TimeSpan.FromSeconds(6));
IObserveable<int> wordGroupCounts = from tick in ticks
    join word in words
        on ticks equals words into wordsInTick
        from count in wordsInTick.Count()
        select count * 10;
wordGroupCounts.Subscribe(c => Console.WriteLine($"Words per minute: {c}"));
```

Результаты не до конца точны, потому что Rx объединяет два элемента, если их время работы пересекается. Это приведет к тому, что некоторые слова будут учитываться несколько раз. Последнее слово в конце одного интервала будет первым словом в начале следующего интервала. В данном случае измерения довольно приблизительны, так что я не слишком беспокоюсь, но вам следует помнить, как именно пересечения влияют на этот тип операций, если, конечно, вам нужны более точные результаты. `Window` или `Buffer` может стать лучшим решением.

Timer

Метод `Observable.Timer` способен создать последовательность, которая выдает ровно один элемент. Он выжидает время, указанное аргументом `TimeSpan`, после чего выдает этот элемент. Он очень похож на `Observable.Interval`, потому что не только принимает схожий аргумент, но и возвращает последовательность того же типа: `IObserveable<long>`. Таким образом, я могу подписаться на этот тип источника почти так же, как и в случае интервальной последовательности, что показано в листинге 11.44.

Листинг 11.44. Единственный элемент с помощью Timer

```
IObserveable<long> src = Observable.Timer(TimeSpan.FromSeconds(1));
src.Subscribe(i => Console.WriteLine($"Event {i} at {DateTime.Now:T}"));
```

Эффект будет таким же, как и при использовании оператора `Interval`, который останавливается после создания первого элемента, так что вы всегда получите нулевое значение. Существуют также перегрузки, которые принимают дополнительный `TimeSpan` и будут повторно генерировать значение, как это делает `Interval`. На самом деле под капотом `Interval` используется `Timer`, так что это просто обертка с более простым API.

Timestamp

В предыдущих двух разделах при выводе сообщений я использовал `DateTime`. `Now` для указания времени создания элементов источниками. Одна потенциальная проблема с этим состоит в том, что в результате мы получаем время, когда обработано сообщение, что не всегда будет точно соответствовать времени получения. Например, если вы использовали `ObserveOn`, чтобы гарантировать, что ваш обработчик всегда работает в потоке пользовательского интерфейса, может наблюдаться значительная задержка между выдачей элемента и обработкой его вашим кодом, происходящая потому, что поток пользовательского интерфейса может быть занят чем-то другим. Вы можете избавиться от этого с помощью оператора `Timestamp`, доступного для любого `IEnumerable<T>`. Листинг 11.45 использует его как альтернативный способ показа времени, когда `Interval` выдает свои элементы.

Листинг 11.45. Элементы с отметками времени

```
IEnumerable<Timestamped<long>> src =  
    Observable.Interval(TimeSpan.FromSeconds(1)).Timestamp();  
src.Subscribe(i => Console.WriteLine(  
    $"Event {i.Value} at {i.Timestamp.ToLocalTime():T}"));
```

Если тип элемента исходного наблюдаемого объекта имеет некоторый тип `T`, этот оператор создаст наблюдаемый объект из элементов `Timestamped<T>`. Он определяет свойство `Value`, содержащее исходное значение из наблюдаемого источника, а также свойство `Timestamp`, указывающее, когда значение прошло через оператор `Timestamp`.



Тип свойства `Timestamp` — это `DateTimeOffset` с нулевым смещением часового пояса (т. е. в формате UTC). Это дает стабильную основу для определения времени, исключая любую возможность перехода на летнее или зимнее время во время работы вашей программы. Однако если вы хотите показать времененную метку конечному пользователю, вам может понадобиться ее скорректировать, поэтому в листинге 11.45 для нее вызывается `ToLocalTime`.

Вы должны применять этот оператор непосредственно к наблюдаемому объекту, который хотите отметить. Запись `src.ObserveOn(sched).Timestamp()` бессмысленна, потому что вы будете помечать элементы уже после того, как они были обработаны планировщиком, переданным в `ObserveOn`. Следует

написать `src.Timestamp().ObserveOn(sched)`, что гарантирует получение временной отметки перед передачей элементов в цепочку обработки, что может привести к задержке.

TimeInterval

Если `Timestamp` записывает текущее выдачи элементов, его близкий родственник `TimeInterval` записывает время между последовательными элементами. Листинг 11.46 использует этот оператор для наблюдаемой последовательности, созданной `Observable.Interval`, так что элементы должны быть расположены достаточно равномерно.

Листинг 11.46. Измерение зазоров

```
IObservable<long> ticks = Observable.Interval(  
    TimeSpan.FromSeconds(0.75));  
IObservable<TimeInterval<long>> timed = ticks.TimeInterval();  
timed.Subscribe(x => Console.WriteLine(  
    $"Event {x.Value} took {x.Interval.TotalSeconds:F3}"));
```

Если элементы `Timestamped<T>`, выданные оператором `Timestamp`, предоставляют свойство `Timestamp`, то элементы `TimeInterval<T>`, созданные оператором `TimeInterval`, определяют свойство `Interval`. Теперь это уже `TimeSpan`, а не `DateTimeOffset`. Я решил вывести количество секунд между каждым элементом с точностью до трех знаков после запятой. Вот что я вижу, когда запускаю его на своем компьютере:

```
Event 0 took 0.760  
Event 1 took 0.757  
Event 2 took 0.743  
Event 3 took 0.751  
Event 4 took 0.749  
Event 5 took 0.750
```

Это интервалы, которые на целых 10 мс отличаются от запрошенных, но это довольно типичное поведение. Windows не является операционной системой реального времени.

Throttle

Оператор `Throttle` позволяет ограничить скорость обработки элементов. Вы передаете аргумент, указывающий минимальный интервал времени между

любыми двумя элементами. Если базовый источник производит элементы быстрее, `Throttle` будет их игнорировать. Если источник работает медленнее, чем указано, `Throttle` пропускает все подряд.

Удивительно (во всяком случае, для меня), что когда источник превышает указанную скорость, `Throttle` пропускает абсолютно все до тех пор, пока скорость не опустится ниже указанного уровня. Таким образом, если вы укажете скорость 10 элементов в секунду, а источник будет выдавать 100, `Throttle` не будет возвращать каждый 10-й элемент. Вместо этого он не будет возвращать вообще ничего, пока источник не замедлится.

Sample

Оператор `Sample` создает элементы из своего ввода в интервале, заданном его аргументом `TimeSpan`, независимо от скорости, с которой наблюдаемый ввод их генерирует. Если базовый источник производит предметы быстрее выбранной частоты, `Sample` отбрасывает предметы, чтобы ограничить частоту. Однако если источник работает медленнее, оператор `Sample` просто повторяет последнее значение, чтобы поддерживать непрерывную выдачу уведомлений.

Timeout

Оператор `Timeout` пропускает через себя все, что выдает наблюдаемый источник, если только этот источник не допускает слишком больших пауз между временем подписки и первым элементом или между двумя последовательными вызовами наблюдателя. Минимально допустимый разрыв указывается с помощью аргумента типа `TimeSpan`. Если в течение этого времени не происходит никакой активности, оператор `Timeout` завершает работу, сообщая об исключении `TimeoutException` в `OnError`.

Операторы окна

Я уже описал операторы `Buffer` и `Window`, но не показал их перегрузки с учетом времени. Помимо возможности указать размер окна и количество пропусков, а также отметить границы окна вспомогательным наблюдаемым источником вы можете задать окна на основе времени.

Если вы просто передадите `TimeSpan`, оба оператора будут разбивать входные данные на смежные окна с заданным интервалом. Это дает значительно бо-

лее простой способ вычислять количество слов в минуту, нежели в листинге 11.43. В листинге 11.47 показано, как добиться того же эффекта с помощью оператора `Buffer`, но используя окно с учетом времени.

Листинг 11.47. Окна с учетом времени с использованием Buffer

```
IObservable<int> wordGroupCounts =  
    from wordGroup in words.Buffer(TimeSpan.FromSeconds(6))  
    select wordGroup.Count * 10;  
wordGroupCounts.Subscribe(c => Console.WriteLine("Words per minute: " + c));
```

Существуют также перегрузки, принимающие как `TimeSpan`, так и `int`, что позволяет вам завершить текущее окно (начав, таким образом, следующее) либо по истечении заданного интервала, либо когда количество элементов превышает пороговое значение. Кроме того, имеются перегрузки, принимающие два аргумента типа `TimeSpan`. Они поддерживают основанный на времени эквивалент комбинации размера окна и количества пропусков. Первый аргумент `TimeSpan` указывает продолжительность окна, а второй — интервал запуска новых окон. Это означает, что окна уже не будут строго смежными — между ними могут оказаться промежутки или же они могут пересекаться. Листинг 11.48 использует это, чтобы проводить более частые вычисления частоты слов, по-прежнему используя шестисекундное окно.

Листинг 11.48. Пересекающиеся окна с учетом времени

```
IObservable<int> wordGroupCounts =  
    from wordGroup in words.Buffer(TimeSpan.FromSeconds(6),  
    TimeSpan.FromSeconds(1))  
select wordGroup.Count * 10;
```

В отличие от формирования фрагментов на основе объединения, которое я показал в листинге 11.43, `Window` и `Buffer` не учитывают элементы дважды, потому что не основаны на концепции пересекающегося времени работы. Они рассматривают прибытие элемента как мгновенное событие, которое либо внутри, либо снаружи любого конкретного окна. Так что примеры, которые я только что продемонстрировал, обеспечат более точную оценку скорости.

Delay

Оператор `Delay` позволяет вам сдвигать наблюдаемый источник по времени. При передаче `TimeSpan` оператор задержит все на указанный срок. Кроме

того, вы можете передать `DateTimeOffset`, указав ему конкретное требуемое время начала воспроизведения его ввода. В качестве альтернативы вы можете передать наблюдаемый объект, и, когда он впервые что-либо выдает или завершается, оператор `Delay` начинает выдавать сохраненные значения.

Независимо от того, как определяется продолжительность сдвига, оператор `Delay` во всех случаях пытается поддерживать одинаковый интервал между входами. Таким образом, если базовый источник производит элемент немедленно, затем другой элемент через три секунды, а затем третий элемент через минуту, наблюдаемый источник, создаваемый `Delay`, будет выдавать элементы с одинаковым интервалом.

Очевидно, что если ваш источник начнет производить предметы особенно стремительно, например по два миллиона в секунду, есть предел точности, с которой `Delay` может выполнить синхронизацию элементов, но он в любом случае сделает все возможное. Пределы точности жестко не заданы. Они будут зависеть от характера используемого вами планировщика и доступной мощности ЦП. Например, если вы используете один из планировщиков на основе пользовательского интерфейса, он будет ограничен доступностью потока пользовательского интерфейса и скоростью, с которой тот может работать. (Как и во всех основанных на времени операторах, `Delay` выберет для вас планировщик по умолчанию, но он имеет перегрузки, которые позволяют вам передать свой.)

DelaySubscription

Оператор `DelaySubscription` предлагает аналогичный оператору `Delay` набор перегрузок, но способ, которым он пытается добиться задержки, отличается. Когда вы подписываетесь на наблюдаемый источник, созданный `Delay`, он сразу же подписывается на базовый источник и начинает буферизацию элементов, перенаправляя каждый элемент только по истечении требуемой задержки. Стратегия, используемая `DelaySubscription`, заключается в том, чтобы попросту отложить подписку на базовый источник, а затем немедленно переслать каждый элемент.

Для холодных источников `DelaySubscription`, как правило, делает то, что нужно, потому что задержка начала работы холодного источника, как правило, сдвигает по времени весь процесс. Но в случае с горячим источником `DelaySubscription` вынудит вас пропустить любые события, которые произошли во время задержки, после чего вы начнете получать события без сдвига по времени.

Оператор `Delay` более надежен — смешая по времени каждый элемент по отдельности, он работает как с горячими, так и с холодными источниками. Тем не менее он вынужден проделывать больше работы, так как буферизует все, что получает за время задержки. В случае нагруженных источников или длительных задержек это может отнять много памяти. Кроме того, попытка воспроизвести исходное время со сдвигом значительно сложнее, чем непосредственная передача элементов. Таким образом, в случаях, когда он применим, `DelaySubscription` работает более эффективно.

Итог

Как вы увидели, реактивные расширения для .NET предоставляют множество функциональных возможностей. Концепция, лежащая в основе Rx, является четко определенной абстракцией для последовательностей элементов, где источник сам решает, когда предоставлять каждый элемент, и связанной абстракцией, представляющей собой подписчика на эту последовательность. Будучи представленными в виде объектов, источники событий и подписчики становятся высокоуровневыми объектами, что означает, что вы можете передавать их в качестве аргументов, сохранять в полях и вообще делать с ними все то, что можно делать с любым другим типом данных .NET. Хотя вы можете проделать все это и с делегатом, события .NET — это уже тот уровень. Более того, Rx предоставляет четко определенный механизм для уведомления подписчика об ошибках, а с этим ни делегаты, ни события не способны справиться на должном уровне. Помимо определения высокоуровневого представления для источников событий Rx определяет всестороннюю реализацию LINQ, поэтому Rx иногда называют LINQ to Events. Фактически он выходит далеко за рамки набора стандартных операторов LINQ, добавляя многочисленные операторы, которые используют и помогают управлять работающим в реальном времени и потенциально чувствительным ко времени окружением, в котором и обитают управляемые событиями системы. Rx также предоставляет различные возможности для связи основных абстракций и абстракций других сред, включая стандартные события .NET, `IEnumerable<T>` и различные асинхронные модели.

ГЛАВА 12

Сборки

До сих пор в этой книге для описания библиотеки или исполняемого файла я использовал термин «компонент». Настало время разобраться, что именно он означает. В .NET правильным термином для программного компонента является термин «сборка», и обычно это файл с расширением `.dll` или `.exe`. Иногда сборка будет разбита на несколько файлов, но даже в этом случае она является неделимой единицей развертывания — вы должны либо предоставить среде выполнения полный доступ ко всей сборке, либо не выпускать ее вообще. Сборки являются важной составляющей системы типов, потому что каждый тип идентифицируется не только по имени и пространству имен, но и по сборке. Сборки обеспечивают поддержку инкапсуляции в большем масштабе, нежели отдельные типы, благодаря внутреннему спецификатору доступности, который действует на уровне сборки.

Среда выполнения предоставляет загрузчик сборок, который автоматически находит и загружает сборки, необходимые программе. Чтобы загрузчик мог найти нужные компоненты, сборки имеют структурированные имена, которые включают информацию о версии, а также могут дополнительно содержать уникальный на глобальном уровне элемент, позволяющий избежать неоднозначности.

В Visual Studio большинство типов проектов C# в диалоговом окне `Create a new project` создают единственную сборку в качестве основного вывода. Кроме того, они часто помещают в выходную папку дополнительные файлы, такие как копии любых сборок, на которые опирается ваш проект и которые не встроены в среду выполнения .NET, а также другие файлы, необходимые вашему приложению. (Например, проект веб-сайта, как правило, должен содержать CSS и файлы сценариев в дополнение к серверному коду.) Но обычно там размещается сборка, которая выступает целью сборки вашего проекта и содержит все типы, определяемые вашим проектом вместе с содержащимся в них кодом.

Анатомия сборки

Сборки используют формат файла Win32 Portable Executable (PE), тот же формат, который исполняемые файлы (EXE) и библиотеки динамических ссылок (DLL) всегда используют в современных версиях Windows. Он «переносимый» в том смысле, что один и тот же базовый формат файла используется в разных архитектурах ЦП¹. Файлы, отличные от формата .NET PE, – это, как правило, архитектурно-зависимые элементы сборок, но сами сборки .NET обычно таковыми не являются. Даже если вы используете .NET Core в Linux или macOS, он все равно будет использовать этот формат на основе Windows – сборки, созданные для .NET Core или .NET Standard, обычно способны работать во всех поддерживаемых операционных системах, так что один и тот же формат файлов можно использовать везде.

Компилятор, как правило C#, создает итоговую сборку с расширением .dll или .exe. Инструменты, которые понимают формат файла PE, расценивают сборку .NET как вполне допустимый, но довольно бесполковый PE-файл. В основном CLR использует PE-файлы в качестве контейнеров для специфичного для .NET формата данных, поэтому по мнению классических инструментов Win32, DLL C# не экспортирует какие-либо API. Как вы помните, C# компилируется в бинарный промежуточный язык (IL), который не исполняется непосредственно. Обычные механизмы Windows для загрузки и запуска кода в исполняемом файле или DLL не будут работать с IL, потому что он может выполняться только посредством CLR. Точно так же .NET определяет свой собственный формат для кодирования метаданных, не используя при этом встроенную в формате PE возможность экспорта точек входа или импорта служб других DLL.

В .NET Core 3.0 или выше сборки .NET не будут иметь расширение .exe. Даже типы проектов, результат сборки которых является исполняемым (например, консольные или WPF-приложения), создают .dll в качестве основного вывода. Дополнительно они генерируют и исполняемый файл, но это уже не сборка .NET. Это просто загрузчик, который запускает среду выполнения, а затем загружает и выполняет основную сборку вашего приложения. По умолчанию тип загрузчика, который вы получаете, зависит от того, на какой ОС вы работаете – например, если вы собираете Windows, вы получите загрузчик Windows .exe, тогда как в Linux это будет исполняемый

¹ Здесь я использую слово «современный» в очень широком смысле – Windows NT добавила поддержку PE в 1993 году.

файл в формате ELF. (Если вы ориентируетесь на .NET Framework, то будут отличия². Поскольку он поддерживается только Windows, ему не нужны загрузчики для разных операционных систем, так что такие проекты создают сборку .NET с расширением .exe, включающим загрузчик.)



Инструменты компиляции с опережением времени (Ahead-of-Time, AoT) в .NET Core могут добавлять собственный исполняемый код к вашим сборкам на более позднем этапе, но с помощью готовых к запуску сборок (так называют результат работы инструментов AoT .NET Core), даже встроенный нативный код загружается и выполняется под контролем CLR и напрямую доступен только для управляемого кода.

Метаданные .NET

Наряду со скомпилированным кодом на IL сборка содержит *метаданные*, которые предоставляют полное описание всех определенных в ней типов, как публичных (public), так и закрытых (private). Чтобы понять IL и превратить его в исполняемый код, CLR нужна полная информация об используемых типах — двоичный формат IL часто ссылается на метаданные содержащейся сборки и без него не имеет смысла. API отражения, о котором пойдет речь в главе 13, делает информацию в этих метаданных доступной вашему коду.

Ресурсы

Вместе с кодом и метаданными вы можете встроить в DLL двоичные ресурсы. Например, клиентские приложения могут включать в себя растровые изображения. Чтобы встроить файл, вы можете добавить его в проект, выбрать его в обозревателе решений, а затем с помощью панели Properties установить его свойство Build Action в Embedded Resource. Это добавит копию всего файла в компонент. Для извлечения ресурса во время выполнения используется метод GetManifestResourceStream класса Assembly, который является частью API отражения, описанного в главе 13. Однако на практике вы, как правило, не используете эту возможность напрямую — большинство приложений

² С подходящими настройками сборки вы можете создавать загрузчики для всех поддерживаемых целевых платформ независимо от того, в какой ОС вы работаете.

используют встроенные ресурсы через механизм локализации, который я опишу позже в этой главе.

В итоге сборка будет содержать полный набор метаданных, описывающих все типы, которые она определяет; она содержит все IL для методов этих типов и при желании может включать любое количество двоичных потоков. Обычно все это упаковано в единственный PE-файл, но бывает, что на этом история не заканчивается.

Многофайловые сборки

.NET давала сборке возможность занимать несколько файлов. Вы могли разделить код и метаданные между несколькими *модулями*, а некоторые двоичные потоки, которые логически встроены в сборку, могли быть, в свою очередь, помещены в отдельные файлы. Эта особенность использовалась достаточно редко, и .NET Core ее не поддерживает. Однако о ней стоит знать, потому что у нее все еще имеются некоторые последствия. В частности, некоторые проектные решения API Reflection (глава 13) останутся непонятными, если вы не знаете об этой функции.

В случае многофайльовой сборки всегда имеется один главный файл, который и представляет сборку. Это PE-файл, и он содержит отдельный элемент метаданных, называемый *манифестом сборки*. Его не следует путать с манифестом в стиле Win32, который содержится в большинстве исполняемых файлов. Манифест сборки — это по сути описание того, что находится в сборке, включая список любых внешних модулей или других внешних файлов; в многомодульной сборке манифест описывает то, в каких файлах определены те или иные типы. При написании кода, который использует типы в сборке напрямую, вам, как правило, не нужно было заботиться о том, разделен ли он на несколько модулей, поскольку загрузчик сверялся с манифестом и автоматически загружал все необходимые модули. Несколько модулей составляли проблему только для кода, который проверял структуру компонента с помощью отражения.

Другие особенности PE

Хотя C# не использует классические механизмы Win32 для представления кода или экспорта API из EXE-файлов и DLL-библиотек, имеется пара старинных возможностей формата PE, которые могут использоваться в сборках.

Работа с ресурсами в стиле Win32

.NET определяет собственный механизм встраивания бинарных ресурсов, на котором основывается встроенный API локализации, поэтому по большей части встроенная поддержка формата файла PE для добавления ресурсов не используется. Ничто не мешает вам помещать классические ресурсы в стиле Win32 в компонент .NET – компилятор C# предлагает различные параметры командной строки, которые способны в этом помочь. Однако не существует API .NET для доступа к этим ресурсам во время выполнения, поэтому обычно используется собственная система ресурсов .NET. Но есть и некоторые исключения.

Система Windows ожидает найти в исполняемых файлах определенные ресурсы. Например, она определяет способ внедрения в качестве неуправляемого ресурса информации о версии. Сборки C# обычно это делают, но вам не нужно указывать ресурс версии явно. Компилятор может создать его за вас, что я показываю в разделе «Версии» на с. 691. Это гарантирует, что если конечный пользователь просмотрит свойства вашей сборки в проводнике Windows, он сможет найти номер версии. (По соглашению, .NET-сборки обычно содержат эту информацию о версии в стиле Win32 независимо от того, предназначены они только для Windows или способны работать на любой платформе.)

Файлы .exe в Windows обычно содержат два дополнительных ресурса Win32. Вам может понадобиться определить пользовательский значок для вашего приложения, чтобы управлять тем, как он отображается на панели задач или в проводнике Windows. Это потребует встраивания иконки методом Win32, потому что проводник не знает, как извлекать ресурсы .NET. Кроме того, если вы пишете классическое настольное приложение Windows или консольное приложение (независимо от того, написано ли оно на .NET), в нем должен содержаться манифест приложения. Без этого Windows считает, что ваше приложение было написано до 2006 года³, и будет изменять или отключать определенные функции в целях обратной совместимости. Манифест также нужен, если вы пишете настольное приложение и хотите, чтобы оно соответствовало определенным требованиям сертификации Microsoft. Этот вид манифеста должен быть встроен в виде ресурса Win32. Повторюсь, вкладка Application на странице свойств проекта поддерживает

³ Год выпуска Windows Vista. Манифесты приложений существовали и до этого, но это была первая версия Windows, которая рассматривала их отсутствие как признак устаревшего кода.

встраивание значка и манифеста, а если вы создаете настольное приложение, Visual Studio настраивает ваш проект для предоставления подходящего манифеста по умолчанию.

Помните, что в .NET Core основной сборкой является `.dll` даже для настольных приложений Windows, а процесс сборки создает отдельный `.exe`, который запускает среду выполнения .NET, после чего загружает саму сборку. Что касается Windows, этот загрузчик и является вашим приложением, поэтому, когда вы ориентируетесь на .NET Core, ресурсы иконки и манифеста окажутся в этой загрузочной сборке. Но если вы ориентируетесь на .NET Framework, отдельного загрузчика не будет, поэтому ресурсы попадут в основную сборку.

Консоль или графический интерфейс?

Windows различает консольные приложения и приложения Windows. Точнее, формату PE требуется, чтобы файл `.exe` указывал подсистему, и во времена Windows NT это позволяло использовать несколько специализаций операционной системы — например, ранние версии включали подсистему POSIX. (Подсистемы ненадолго возродились в 2017 году вместе с подсистемой Linux для Windows, которая позволяет запускать исполняемые файлы Linux непосредственно в Windows 10. Но в 2019 году Microsoft для улучшения совместимости в поддержке Linux переключилась с подсистемы на специализированную облегченную виртуальную машину.) В наши дни PE-файлы предназначены для одной из трех подсистем, одна из которых предназначена для драйверов устройств в режиме ядра. Два пользовательских режима, которые используются сегодня, — это графический интерфейс Windows (GUI) и консольное приложение Windows. Принципиальное отличие заключается в том, что при запуске последнего Windows будет показывать окно консоли (или, если вы запускаете его из командной строки, будет использовано существующее), а приложение с графическим интерфейсом Windows не получает окна консоли.

Вы можете выбрать одну из этих подсистем на странице свойств приложения проекта, используя раскрывающийся список `Output type`. В нем имеются эти два пункта: `Windows Application` и `Console Application`. (Кроме того, там есть `Class Library`, что создает библиотеку DLL, но, поскольку подсистема определяется при запуске процесса, не имеет значения, ориентируется библиотека DLL на консоль Windows или подсистему пользовательского интерфейса. Настройка `Class Library` всегда нацелена на первое.) Если вы ориентируетесь на .NET

Framework, этот параметр подсистемы применяется к файлу .exe, который создается как основная сборка вашего приложения. В более новых версиях .NET он будет применяться к загрузчику в формате .exe. (Так повелось, что он также будет применяться к основной сборке .dll, которую загружает загрузчик, но это уже не имеет никакого значения, поскольку подсистема определяется .exe, для которого запущен процесс.)

Определение типа

Первое, что следует усвоить разработчику C# относительно сборок — это то, что они являются частью определения типа. Любой написанный класс в итоге окажется в сборке. Когда вы используете тип из библиотеки классов .NET или какой-либо другой, вашему проекту потребуется ссылка на сборку, которая содержит тип, и только после этого вы сможете его использовать.

Это не всегда очевидно при использовании системных типов. Система сборки автоматически добавляет ссылки на различные сборки библиотеки классов .NET, поэтому по большей части вам не нужно добавлять ссылку, прежде чем использовать тип библиотеки классов .NET. И, поскольку вы обычно не ссылаетесь на сборку типа явно, в исходном коде не всегда очевидно, что указание сборки является обязательным для точного определения типа. Но несмотря на то, что сборка не появляется в коде явно, она остается частью определения типа, потому что ничто не мешает вам или кому-либо еще определять новые типы, имена которых совпадают с именами существующих. Например, в своем проекте вы можете определить класс с именем `System.String`. Это плохая идея, и компилятор предупредит вас, что этот шаг вносит двусмысленность, но при этом не остановит вас. И хотя ваш класс будет иметь точно такое же полное имя, как и встроенный строковый тип, компилятор и среда выполнения смогут различать эти типы.

Всякий раз, когда вы используете тип явно по имени (например, в объявлении переменной или параметра) или неявно через выражение, компилятор C# точно знает, на какой именно тип вы ссылаетесь, т. е. какая сборка его определяет. Таким образом, он способен различать `System.String`, встроенный в .NET, и `System.String`, бездумно определенный в вашем собственном компоненте. Правила области видимости C# подскажут, что явная ссылка на `System.String` идентифицирует ту, которую вы определили в своем собственном проекте, потому что локальные типы фактически скрывают типы с идентичными именами во внешних сборках. Если вы используете ключевое

слово `string`, оно всегда будет относиться к встроенному типу. Встроенный тип также будет использоваться при работе со строковым литералом или вызове API, который возвращает строку. Листинг 12.1 определяет собственный `System.String`, а затем использует обобщенный метод, который отображает тип и имя сборки для статического типа любого передаваемого аргумента. (При этом используется API отражения, который описан в главе 13.)

Листинг 12.1. Какого типа строка?

```
using System;

// Никогда так не делайте!
namespace System
{
    public class String
    {
    }

    class Program
    {
        static void Main(string[] args)
        {
            System.String s = null;
            ShowStaticTypeNameAndAssembly(s);
            string s2 = null;
            ShowStaticTypeNameAndAssembly(s2);
            ShowStaticTypeNameAndAssembly("String literal");
            ShowStaticTypeNameAndAssembly(
                Environment.OSVersion.VersionString);
        }

        static void ShowStaticTypeNameAndAssembly<T>(T item)
        {
            Type t = typeof(T);
            Console.WriteLine(
                $"Type: {t.FullName}. Assembly {t.Assembly.FullName}.");
        }
    }
}
```

Метод `Main` в этом примере пробует поработать с каждым описанным типом строк и выдает следующее:

```
Type: System.String. Assembly MyApp, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null.
```

```
Type: System.String. Assembly System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e.
Type: System.String. Assembly System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e.
Type: System.String. Assembly System.Private.CoreLib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e.
```

Явное использование `System.String` привело к использованию моего типа, а все остальные использовали системный. Это демонстрирует, что компилятор C# вполне способен разобраться с несколькими типами с одинаковыми именами. Это также показывает, что и IL способен их различать. Бинарный формат IL гарантирует, что каждая ссылка на тип указывает на содержащую сборку. Но возможность создавать и использовать несколько типов с одинаковыми именами не означает, что вам следует это делать. Поскольку в C# вы обычно явно не указываете содержащую сборку, особенно плохой идеей будет добавление бессмысленных коллизий при определении, скажем, собственного класса `System.String`. (В крайнем случае, если это действительно нужно, вы можете разрешить такого рода коллизии — подробности во врезке «Внешние псевдонимы» на с. 676, — но лучше их все же избегать.)

Кстати, если вы запустите листинг 12.1 в .NET Framework, то вместо `System.Private.CoreLib` вы увидите `mscorlib`. В .NET Core многие типы из библиотек классов поменяли прописку. Вы можете спросить, как это работает в .NET Standard, который позволяет написать одну DLL для работы и в .NET Framework, и в .NET Core. Как компонент .NET Standard может правильно определить тип, который находится в разных сборках для разных целевых платформ? Ответ заключается в том, что .NET имеет функцию *переадресации типов*, которая позволяет перенаправлять ссылки на типы в другую сборку прямо во время выполнения. (Переадресация типов — это просто атрибут уровня сборки, который описывает, где можно найти определение реального типа. Атрибуты являются темой главы 14.) Стандартные компоненты .NET не ссылаются ни на `mscorlib`, ни на `System.Private.CoreLib` — они собираются так, будто типы библиотек классов определены в сборке `netstandard`. Каждая среда выполнения .NET содержит реализацию `netstandard`, которая переадресует нужные типы во время выполнения. На самом деле даже код, созданный непосредственно для .NET Core, часто в итоге занимается переадресацией. Если вы изучите скомпилированный вывод, то обнаружите, что он ожидает найти большинство типов библиотек классов .NET в сборке с именем `System.Runtime`,

и только переадресация типов позволяет использовать типы, определенные в `System.Private.CoreLib`.

Если иметь несколько типов с одинаковым именем — плохая затея, почему .NET вообще делает это возможным? На самом деле поддержка конфликтных имен была не самоцелью, а всего лишь побочным эффектом того, что .NET делает сборку частью типа. Сборка должна быть частью определения типа, чтобы среда выполнения могла понять, какую сборку загрузить во время выполнения, когда вы впервые используете какую-либо функцию этого типа.

ВНЕШНИЕ ПСЕВДОНИМЫ

Когда в области видимости находятся несколько типов с одним и тем же именем, C# обычно использует тип из ближайшей области, поэтому локально определенный `System.String` способен скрыть встроенный тип с тем же именем. В первую очередь неразумно вообще допускать подобный конфликт имен, но иногда вы можете столкнуться с этой проблемой, если внешние библиотеки, от которых зависит ваша сборка, нарушили правила именования. Если это ваш случай, то C# содержит механизм, который позволяет вам указать нужную сборку. Вы можете определить внешний псевдоним.

В главе 1 я показал псевдонимы типов, определяемые с помощью ключевого слова `using`. Они упрощают обращение к типам, которые имеют одинаковое имя, но расположены в разных пространствах имен. Внешний псевдоним позволяет различать типы с одинаковым полным именем в разных сборках.

Чтобы определить внешний псевдоним, разверните список `Dependencies` в `Solution Explorer`, а затем разверните раздел `Projects or Assemblies` и выберите ссылку. (Эту технику нельзя использовать для ссылок, полученных через NuGet.) После этого вы можете задать псевдоним для этой ссылки на панели `Properties`. Если для одной сборки определен псевдоним `A1`, а для другой — `A2`, то вы можете заявить о желании использовать эти псевдонимы, поместив в начало файла C# следующие строки:

```
extern alias A1;  
extern alias A2;
```

Это даст возможность определять имена типов с помощью `A1::` или `A2::`, за которыми должно следовать полное имя. Это говорит компилятору, что вам нужны типы, определенные сборкой (или сборками), связанной с этим псевдонимом, даже если какой-то другой тип с таким же именем находится в области видимости.

Загрузка сборок

Вас могли насторожить мои слова о том, что система сборки автоматически добавляет ссылки на все компоненты библиотеки классов .NET, доступные в вашей целевой среде. У вас мог возникнуть вопрос, как можно удалить некоторые из них для улучшения производительности. Что касается времени выполнения, беспокоиться об этом не нужно. Компилятор C# по сути игнорирует любые ссылки на встроенные сборки, которые ваш проект никогда не использует, поэтому опасности загрузки ненужных вам DLL нет. (Однако стоит все же удалить ссылки на неиспользуемые компоненты, которые не являются встроенными, чтобы избежать копирования ненужных библиотек DLL при развертывании приложения — нет смысла делать его больше, чем нужно. Но неиспользуемые ссылки на библиотеки DLL, которые уже установлены как часть .NET, не несут дополнительных затрат.)

Даже если C# не удалит неиспользуемые ссылки во время компиляции, риска ненужной загрузки неиспользуемых библиотек DLL все равно не будет. CLR не пытается загружать сборки, пока они не понадобятся вашему приложению. Большинство приложений во время выполнения не используют весь возможный код, так что довольно часто значительная часть кода в вашем приложении никогда не запускается. Ваша программа может даже завершить свою работу, оставив неиспользованными целые классы — например, те, которые включаются в работу только при возникновении необычной ошибки. Если единственное место, где вы используете конкретную сборку, находится внутри метода такого класса, эта сборка загружаться не будет.

CLR по своему усмотрению решает, что именно означает «использование» определенной сборки. Если метод содержит код, который ссылается на определенный тип (например, объявляет переменную этого типа или содержит выражения, которые неявно его используют), то CLR может посчитать этот тип использующимся при первом запуске этого метода, даже если выполнение не дойдет до части, которая действительно его использует. Рассмотрим листинг 12.2.

В зависимости от своего аргумента эта функция либо возвращает объект, предоставленный `StringComparer` из библиотеки классов .NET, либо создает новый объект типа `MyCustomComparer`. Тип `StringComparer` определяется в той же сборке, что и базовые типы, такие как `int` и `string`, поэтому он будет загружен при запуске программы. Но предположим, что другой тип, `MyCustomComparer`, был определен в отдельной сборке под названием

`ComparerLib`. Очевидно, что если метод `GetComparer` вызывается с аргументом `false`, CLR потребуется загрузить `ComparerLib`, если это еще не было сделано. Но вот в чем загвоздка. Он, скорее всего, загрузит `ComparerLib` при первом же вызове этого метода, даже если аргумент равен `true`. Чтобы JIT мог скомпилировать метод `GetComparer`, CLR потребуется доступ к определению типа `MyCustomComparer`. Как минимум это нужно для проверки того, действительно ли тип имеет конструктор с нулевым аргументом. (Очевидно, что в таком случае листинг 12.2 не будет компилироваться, но возможно, что код был скомпилирован с использованием другой версии `ComparerLib`, нежели та, что доступна во время выполнения.) Работа JIT-компилятора является деталью реализации, поэтому она не полностью документирована и может меняться от версии к версии, но похоже, что она обрабатывает по одному методу за раз. Так что простого вызова этого метода, по всей видимости, будет достаточно для загрузки сборки `ComparerLib`.

Листинг 12.2. Загрузка типа и условное исполнение

```
static IComparer<string> GetComparer(bool useStandardOrdering)
{
    if (useStandardOrdering)
    {
        return StringComparer.CurrentCulture;
    }
    else
    {
        return new MyCustomComparer();
    }
}
```

Это поднимает вопрос о том, как .NET находит сборки. Если сборки могут быть загружены неявно в результате выполнения метода, у нас не всегда будет возможность сообщить среде выполнения их местоположение. Так что в .NET для этого имеется специальный механизм.

Разрешение сборок

Когда среди выполнения требуется загрузить сборку, она проходит процесс, называемый разрешением сборки. В некоторых случаях вы просите .NET загружать конкретную сборку (например, при первом запуске приложения), но большинство из них загружается неявно. Точный механизм зависит от нескольких факторов: ориентируетесь ли вы на .NET Core или на более старую .NET Framework и, если первое, является ли ваше приложение автономным.

Среда .NET Core поддерживает два варианта развертывания приложений: автономный и зависящий от окружения. Когда вы публикуете автономное приложение, оно включает в себя полную копию .NET Core — полный CLR и все встроенные сборки. В листинге 12.3 показана командная строка для создания такого приложения — если вы запустите ее из папки, содержащей файл .csproj, она скомпилирует проект, а затем создаст папку publish, содержащую ваш скомпилированный код и полную копию подходящей версии .NET Core. (Версия будет зависеть от настроек целевой платформы вашего проекта. Как правило, в файле проекта указываются основная и вспомогательная версии, например netcoreapp3.0, после чего SDK добавляет последнюю версию набора изменений, установленную на вашем компьютере. Доступные версии будут зависеть от того, какие версии .NET Core SDK вы установили.) Ключ -r указывает платформу, для которой необходимо создать сборку: CLR для Linux обязательно в чем-то отличается от такового для Windows, то же касается и версии для macOS. А в случае Windows и Linux существуют версии для процессоров с архитектурой Intel (как 32-разрядных, так и 64-разрядных), а также ARM. Система сборки должна точно знать, какую из них копировать. В листинге 12.3 выбирается среда выполнения для Windows, работающая на 64-битных процессорах Intel.

Листинг 12.3. Публикация автономного приложения

```
dotnet publish -c Release -r win-x64 --self-contained true
```

Когда вы собираете его таким образом, сборки разрешаются довольно просто, потому что все — собственные сборки вашего приложения, любые внешние библиотеки, от которых вы зависите, все системные сборки, встроенные в .NET, и сам CLR — оказываются в одной папке. (На момент написания это составляло около 66 МБ для простого консольного приложения «Hello world» на .NET Core 3.0 для этой же целевой архитектуры.)

Есть два основных преимущества автономного развертывания. Во-первых, нет необходимости устанавливать .NET на целевые машины — приложение может быть запущено напрямую, поскольку содержит собственную копию .NET. Во-вторых, вы точно знаете, с какой версией .NET и с какими версиями всех библиотек DLL вы работаете. Microsoft делает все возможное, чтобы обеспечить обратную совместимость с новыми выпусками, но иногда могут происходить критические изменения. Автономное развертывание может оказаться единственным выходом, если вы обнаружили, что ваше приложение перестало работать после обновления .NET Core. При автономном

развертывании, если только приложение не требует от CLR осуществлять поиск в другом месте, все будет загружаться из папки приложения, включая все сборки, встроенные в .NET.

Но что, если вы не хотите помещать полную копию .NET Core в итоговую сборку? Поведение сборки приложений по умолчанию — это создание зависимого от окружения исполняемого файла. (Существует вариант под названием «зависимое от окружения развертывание», который является почти тем же самым, за исключением отсутствия в нем исполняемого файла загрузчика. Для запуска такого приложения вам потребуется через инструмент командной строки `dotnet` запустить среду выполнения, после чего уже она запустит ваше приложение. До версии 3.0 .NET Core по умолчанию использовал такое развертывание. Его преимущество — полная независимость от платформы; загрузчик при зависящем от окружения исполняемом развертывании всегда зависит от ОС. Но это менее удобно — вы не можете запустить результат сборки без инструмента `dotnet`.) В этом случае ваш код использует подходящую версию .NET Core, уже установленную на компьютере. Результат будет содержать вашу собственную сборку приложения и, возможно, сборки, от которых зависит ваше приложение, но не будет содержать библиотеки, встроенные в .NET.

Зависимые от окружения приложения в обязательном порядке используют более сложный механизм разрешения, нежели автономные. Когда такое приложение запускается, оно сначала решит, какую именно версию .NET Core следует запустить. Это не обязательно будет версия, для которой было собрано ваше приложение, и есть различные опции для настройки этого выбора. По умолчанию, если доступна та же версия `Major.Minor`, использоваться будет именно она. Например, если зависящее от окружения приложение, созданное для .NET Core 2.2, запускается на компьютере с установленными версиями .NET Core 2.1.12, 2.2.6 и 3.0.0, оно выберет 2.2.6. В случаях, когда такое совпадение недоступно, но есть совпадение с основным номером версии, приложение обычно использует его. Например, если приложение предназначено для 2.1, а на машине установлена только версия 2.2.6, оно будет работать на ней. Также возможен запуск с более высоким номером основной версии, чем использовался при сборке приложения (например, сборка для 2.1, но запуск на 3.0), но только при явном запросе этого через конфигурацию.

Выбранная версия среды выполнения определяет не только CLR, но также и сборки, составляющие части библиотеки классов, встроенной в .NET.

Обычно вы можете найти все установленные версии среды выполнения в папке `C:\Program Files\dotnet\shared\Microsoft.NETCore.App\` в Windows или в `/usr/share/dotnet/shared/Microsoft.NETCore.app` в Linux, в которой дополнительно будут содержаться подпапки с версиями, такие как `3.0.0`. (Не следует полагаться на эти пути — в будущих версиях .NET файлы могут быть перемещены.) Процесс разрешения сборки будет осуществлять поиск в папке, содержащей номер версии, и именно так приложения, зависящие от окружения, используют встроенные сборки .NET.

Если вы покопаетесь в этих папках, в `shared` вы найдете и другие подпапки, например `Microsoft.AspNetCore.App`. Похоже, что этот механизм предназначен не только для встроенных в .NET файлов библиотеки классов .NET, но также для установки сборок для целых платформ. Приложения .NET Core заявляют, что они используют определенную среду приложений. (Инструменты сборки автоматически создают файл с расширением `.runtimeconfig.json` в выходных данных сборки, объявляя используемую платформу. В консольных приложениях указывается `Microsoft.NETCore.App`, тогда как в веб-приложениях — `Microsoft.AspNetCore.App`.) Это позволяет приложениям, ориентированным на конкретные платформы Microsoft, не включать полную копию всех DLL-библиотек среды, даже если она не является частью самой .NET Core.

Если вы установите чистую среду выполнения .NET Core, вы получите только `Microsoft.NETCore.App` и ни одной платформы приложений. Таким образом, приложения, предназначенные для таких платформ, как ASP.NET Core или WPF, не смогут работать, если они собраны по умолчанию, поскольку это предполагает, что эти платформы должны быть предварительно установлены на целевых машинах и процесс разрешения сборки не сможет найти компоненты, зависящие от окружения. .NET Core SDK устанавливает эти дополнительные компоненты, поэтому вы не столкнетесь с этой проблемой на своем рабочем компьютере, но, возможно, увидите ее при развертывании во время выполнения. Вы можете указать инструментам сборки, чтобы они включали компоненты платформы, но обычно этого не требуется. Если вы запускаете свое приложение в публичной облачной службе, такой как `Azure`, там обычно предварительно установлены соответствующие компоненты окружения, поэтому на практике вы обычно сталкиваетесь с такой ситуацией, только если настраиваете сервер самостоятельно или при развертывании приложений для настольных компьютеров. Для этих случаев Microsoft предлагает установщики для среды выполнения

.NET Core, которые также включают компоненты для веб-платформ или настольных систем.

Папка `shared` в пакете установки `dotnet` — это не то место, где стоит вручную что-то менять. Она предназначена только для собственных платформ Microsoft. Однако при желании можно установить дополнительные общесистемные компоненты, поскольку .NET Core также поддерживает нечто, называемое хранилищем пакетов времени выполнения. Это дополнительный каталог, структурированный почти так же, как только что описанная папка `shared`. Вы можете создать подходящий макет каталога с помощью команды `dotnet store`, и если вы установите переменную окружения `DOT_NET_SHARED_STORE`, CLR сможет осуществлять в ней поиск во время разрешения сборки. Это позволяет вам использовать ту же хитрость, что и в средах Microsoft: вы можете создавать приложения, которые зависят от набора компонентов, без необходимости включать их в сборку, если вам удалось организовать предварительную установку этих компонентов на целевой машине.

Помимо поиска в этих двух местах расположения общих платформ во время разрешения сборки CLR будет осуществлять поиск и в собственном каталоге приложения, так же как и в случае автономного приложения. Кроме того, CLR содержит механизмы, позволяющие применять обновления. Например, в случае Windows Microsoft может выпускать критические обновления компонентов .NET Core через Центр обновления Windows.

Говоря в общем, основной процесс разрешения сборки для приложений, зависящих от платформы, заключается в том, что неявная загрузка сборки происходит либо из каталога приложения, либо из общего набора компонентов, установленных на машине. (Это также верно для приложений, работающих на более старой платформе .NET Framework, хотя механизмы будут немного отличаться. В ней есть нечто, называемое глобальный кэш сборок (Global Assembly Cache, GAC), который эффективно объединяет функциональность, предоставляемую обоими общими хранилищами .NET Core. Он не такой гибкий, потому что местоположение хранилища фиксировано. То, как .NET Core использует переменную среды, открывает возможность использования разных общих хранилищ для разных приложений.)

Явная загрузка

Хотя CLR умеет загружать сборки автоматически, их можно загрузить и явно. Например, если вы создаете приложение, которое поддерживает

подключаемые модули, во время разработки вы не будете точно знать, какие компоненты следует загружать во время выполнения. Весь смысл системы подключаемых модулей в том, что она расширяемая, поэтому вы, скорее всего, захотите загрузить все библиотеки DLL в отдельную папку. (Вам придется использовать отражение, чтобы обнаружить и использовать типы из этих DLL, что описано в главе 13.)

Если известен полный путь сборки, ее загрузка очень проста: вы вызываете статический метод `LoadFrom` класса `Assembly`, передавая ему путь к файлу. Путь может задаваться относительно текущего каталога или быть абсолютным. Этот статический метод возвращает экземпляр класса `Assembly`, который является частью API отражения. Он предоставляет способы обнаружения и использования типов, определенных сборкой.

Иногда вам может понадобиться загрузить компонент явно (например, использовать его с помощью отражения), не указывая путь. Скажем, вы можете загрузить определенную сборку из библиотеки классов .NET. Никогда не следует жестко задавать местоположение системного компонента — они имеют тенденцию перемещаться из одной версии .NET в другую. Если в вашем проекте есть ссылка на соответствующую сборку и вы знаете имя определяемого ею типа, вы можете написать `typeof (TheType).Assembly`. Но если это не вариант, то следует использовать метод `Assembly.Load`, передав ему имя сборки.



В некоторых случаях динамическая загрузка запрещена. Например, приложения, созданные для Windows 10 с использованием UWP, установленные из Microsoft Store, могут запускать код только из компонентов, которые поставляются как часть приложения. Это связано с тем, что Microsoft всесторонне тестирует приложения магазина, пытаясь предотвратить проблемы с безопасностью и стабильностью работы, для чего необходим доступ ко всему коду вашего приложения. Возможность загрузки и запуска внешнего кода будет препятствовать этим проверкам.

`Assembly.Load` использует точно такой же механизм, что и неявно запускаемая загрузка. Так вы можете ссылаться либо на компонент, который установили вместе с вашим приложением, либо на системный компонент. В любом случае вы должны указать полное имя, которое должно содержать информацию об имени и версии, например `ComparerLib, Version=1.0.0.0, Culture=neutral, PublicKeyToken =null`.

Версия CLR .NET Framework запоминает, какие сборки были загружены с помощью `LoadFrom`. Если сборка, загруженная таким образом, запускает неявную загрузку других сборок, CLR будет искать их там, откуда эта сборка была загружена. Это означает, что если ваше приложение хранит подключаемые модули в отдельной папке, в которую CLR обычно не заглядывает, эти модули могут устанавливать другие компоненты, от которых они зависят, расположенные в той же папке. CLR найдет их без того, чтобы дополнительно вызывать `LoadFrom`, даже если при обычной неявно инициируемой загрузке он бы не стал заглядывать в эту папку. Однако в .NET Core такое поведение не поддерживается. В этой библиотеке имеется другой механизм для поддержки сценариев с загружаемыми модулями.

Изоляция и загружаемые модули с использованием `AssemblyLoadContext`

В .NET Core появился тип с именем `AssemblyLoadContext`. Он обеспечивает определенный уровень изоляции между группами сборок в рамках одного приложения. Это решает проблему, которая может возникнуть в приложениях, поддерживающих модель загружаемых модулей⁴.

Если такой модуль зависит от какого-либо компонента, который, в свою очередь, используется основным приложением, но каждому из них нужна своя версия, это может вызвать проблемы, если использовать простые механизмы, описанные в предыдущем разделе. Обычно среда выполнения .NET унифицирует эти ссылки, загружая только одну версию. Во всех случаях, когда типы в этом совместно используемом компоненте являются частью интерфейса загружаемого модуля, это именно то, что вам нужно: если приложению требуются загружаемые модули для реализации интерфейса, который опирается на типы, например из библиотеки `Newtonsoft.Json`, важно, чтобы и приложения, и модули согласованно использовали определенную версию библиотеки.

Но унификация может вызвать проблемы с компонентами, представляющими собой детали реализации, а не часть API между приложением и его модулями. Если базовое приложение использует, скажем, версию `2.2 Microsoft.Extensions.Logging` для внутреннего использования, а плагин использует

⁴ Это недоступно в .NET Framework, а также отсутствует в любой текущей версии .NET Standard на момент написания. Обычно в .NET Framework изоляцией управляли домены приложений, более старый механизм, который не поддерживается в .NET Core.

версию 3.0 того же компонента, то нет особой необходимости унифицировать его до использования одного варианта во время выполнения — в подключении отдельных версий для приложения и модуля не будет никакого вреда. Унификация может даже вызвать проблемы: принудительное использование модулем версии 2.2 вызовет исключения во время выполнения, если он попытается использовать функции, присутствующие только в версии 3.0. Принудительное использование приложением версии 3.0 также способно привести к ошибкам, поскольку значительные изменения номера версии часто подразумевают критические изменения.

Чтобы избежать подобного, вы можете добавить пользовательские контексты загрузки сборки. Можно написать класс, производный от `AssemblyLoadContext`, и для каждой из реализаций среды выполнения .NET создает соответствующий контекст загрузки, который поддерживает загрузку разных версий сборок, возможно, уже загруженных приложением. Точную политику вы можете задать, перегрузив метод `Load`, как показано в листинге 12.4.

Листинг 12.4. Пользовательский `AssemblyLoadContext` для загружаемых модулей

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Loader;

namespace HostApp
{
    public class PlugInLoadContext : AssemblyLoadContext
    {
        private readonly AssemblyDependencyResolver _resolver;
        private readonly ICollection<string> _plugInApiAssemblyNames;

        public PlugInLoadContext(
            string pluginPath,
            ICollection<string> plugInApiAssemblies)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
            _plugInApiAssemblyNames = plugInApiAssemblies;
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            if (!_plugInApiAssemblyNames.Contains(assemblyName.Name))
            {
                string assemblyPath =
                    _resolver.ResolveAssemblyToPath(assemblyName);
            }
        }
    }
}
```

```
        if (assemblyPath != null)
    {
        return LoadFromAssemblyPath(assemblyPath);
    }
}

return AssemblyLoadContext.Default.LoadFromAssemblyName(
    assemblyName);
}
}
```

Он принимает расположение DLL загружаемого модуля, а также список имен любых специальных сборок, в которых он должен использовать ту же версию, что и базовое приложение. (Это будет включать типы, определяющие интерфейсы, используемые в интерфейсе вашего модуля. Вам не нужно включать сборки, которые составляют сам .NET, — они всегда унифицированы, даже при использовании пользовательских контекстов загрузки.) Среда выполнения будет вызывать метод `Load` этого класса каждый раз, когда сборка загружается в этом контексте. Данный код проверяет, является ли загружаемая сборка специальной, т. е. одной из тех, что должны быть общими для модулей и базового приложения. Если это не так, то поиск собственной версии этой сборки осуществляется уже в папке модуля. В случаях, когда пример не использует сборку из папки модуля (либо потому, что модуль не предоставил эту конкретную сборку, либо из-за того, что она является специальной), этот контекст полагается на `AssemblyLoadContext.Default`, что означает, что в данном случае базовое приложение и модуль используют одни и те же сборки. Листинг 12.5 показывает это в действии.

Листинг 12.5. Использование контекста загрузки модуля

```
Assembly[] plugInApiAssemblies =
{
    typeof(IPPlugIn).Assembly,
    typeof(JsonReader).Assembly
};
var plugInAssemblyNames = new HashSet<string>(
    plugInApiAssemblies.Select(a => a.GetName().Name));
var ctx = new PlugInLoadContext(plugInDllPath, plugInAssemblyNames);
Assembly plugInAssembly = ctx.LoadFromAssemblyPath(plugInDllPath);
```

Код создает список сборок, которые модуль и приложение должны совместно использовать, и вместе с путем к DLL модуля передает их имена в контекст модуля. Любые библиотеки DLL, от которых зависит модуль и которые

скопированы в ту же папку, что и сам модуль, будут загружены, если только они не находятся в этом списке. В таком случае модуль будет использовать ту же сборку, что и само базовое приложение.

Имена сборок

Названия сборок подчиняются определенным правилам. Они всегда включают простое имя, т. е. имя, под которым вы обычно ссылаетесь на DLL, например `MyLibrary` или `System.Runtime`. Обычно они идентичны имени файла, но без расширения. Технически это не обязательно, но механизм разрешения сборки предполагает, что это так⁵. Имена сборок всегда включают номер версии. Есть также несколько дополнительных компонентов, включая маркер открытого ключа, — строку шестнадцатеричных цифр, которая нужна, если вам требуется уникальное имя.

АСИММЕТРИЧНОЕ ШИФРОВАНИЕ

Если вы не знакомы с асимметричным шифрованием, то здесь не найдете исчерпывающей информации. Я дам лишь краткий обзор. Строгие имена используют алгоритм шифрования RSA, который работает с парой ключей: открытым и закрытым. Сообщения, зашифрованные открытым ключом, могут быть расшифрованы только с использованием закрытого ключа, и наоборот. .NET использует это для формирования цифровой подписи сборки: чтобы подписать сборку, вы вычисляете хеш ее содержимого, а затем шифруете его с помощью закрытого ключа. Затем эта подпись копируется в сборку, и ее достоверность может проверить любой, у кого есть доступ к открытому ключу, — можно самостоятельно вычислить хеш содержимого сборки, расшифровать вашу подпись с помощью открытого ключа, и, если результаты различаются, подпись недействительна. Это подразумевает, что либо она не была создана владельцем закрытого ключа, либо файл был изменен с момента создания подписи и поэтому является подозрительным. Математика шифрования предполагает, что практически невозможно создать подпись, выглядящую действительной, без доступа к закрытому ключу, а также изменить сборку без изменения хеша. А в криптографии «практически невозможно» означает «теоретически возможно, но слишком затратно в плане вычислений, чтобы иметь хоть какой-то смысл, если только не произойдет какого-то серьезного внезапного прорыва в теории чисел или, возможно, квантовых вычислениях, что сделает большинство современных крипtosистем бесполезными».

⁵ Если вы используете `Assembly.LoadFrom`, CLR не волнует, соответствует ли имя файла простому имени.

Строгие имена

Если имя сборки включает в себя маркер открытого ключа, оно называется строгим. Microsoft рекомендует, чтобы у любого компонента .NET, публикуемого для совместного использования (например, доступного через NuGet), было строгое имя. Цель строгого именования состоит в том, чтобы сделать имя сборки уникальным. В связи с этим вы можете задаться вопросом, почему .NET не просто не использовать глобальный уникальный идентификатор (Globally Unique Identifier, GUID)? Ответ состоит в том, что исторически строгие имена служили и для другой цели: они были разработаны, чтобы обеспечить некоторую степень уверенности в том, что целостность сборки не была нарушена. Ранние версии .NET проверяли сборки со строгими именами на наличие искажений во время выполнения, но эти проверки были прекращены, поскольку приводили к значительному потреблению ресурсов во время выполнения, часто не давая никаких преимуществ. Документация Microsoft в настоящее время явно запрещает рассматривать строгие имена как функцию безопасности. Однако для того, чтобы понимать и использовать строгие имена, вам необходимо знать, для какой цели они предназначались изначально.

Как видно из названия, маркер открытого ключа имени сборки связан с криптографией. Это шестнадцатеричное представление 64-битного хеша открытого ключа. Сборки со строгими именами должны содержать копию полного открытого ключа, из которого был создан хеш. Формат файла сборки также обеспечивает пространство для цифровой подписи, созданной с помощью соответствующего закрытого ключа.



Подпись, связанная со строгим именем, не зависит от **Authenticode**, традиционного механизма подписи кода в Windows. Они служат разным целям. **Authenticode** обеспечивает отслеживаемость, так как его открытый ключ обернут в сертификат, который сообщает, откуда пришел код. В случае маркера открытого ключа строгого имени вы получаете только число, поэтому, если вы случайно не узнаете, кому принадлежит этот маркер, он ничего вам не скажет. **Authenticode** позволяет получить ответ на вопрос: «Откуда взялся этот компонент?» Маркер открытого ключа позволяет вам сказать: «Это нужный мне компонент». Обычно один и тот же компонент .NET использует оба механизма.

КЛЮЧИ, СТРОГИЕ ИМЕНА И ОТКРЫТАЯ ПОДПИСЬ

Существует три популярных подхода в работе со строгими именами. Самый простой — использовать настоящие имена в процессе разработки, а также копировать открытый и закрытый ключи на машины всех разработчиков, чтобы они могли каждый раз подписывать сборки. Этот подход жизнеспособен только в том случае, когда вам не нужно хранить закрытый ключ в тайне, поскольку разработчики способны его легко скомпрометировать, случайно или намеренно. Поскольку строгие имена больше не являются фактором безопасности, в этом нет ничего страшного. Тем не менее некоторые организации в соответствии со своей политикой предпочитают хранить свои закрытые ключи при себе, поэтому вы можете столкнуться с другими подходами.

Например, использовать совершенно другой набор ключей во время разработки, переключаясь на реальное имя только для ответственных финальных сборок. Это устраняет необходимость всем разработчикам иметь копию настоящего закрытого ключа, но может привести к путанице, так как разработчики могут оказаться в ситуации с двумя наборами компонентов: один с именами разработки, а другой — с реальными.

Третий подход заключается в использовании реальных имен во всех случаях, но вместо подписи каждой сборки можно просто заполнять нулями часть файла, зарезервированную под подпись. .NET Core называет это *открытой подписью*, и это скорее соглашение, нежели функция: подпись работает лишь потому, что .NET Core CLR никогда не проверяет подписи строгого именованных сборок. (.NET Framework по-прежнему проверяет подписи в ряде случаев. Например, для установки в GAC сборка должна иметь строгое имя с действительной подписью. Там имеется чуть более сложный механизм, называемый отсроченной подписью. Он заставит вас сделать пару лишних телодвижений для достижения того же эффекта: разработчики могут компилировать сборки, которые имеют действительные строгие имена, без необходимости генерировать подписи.)

Вы можете создать файл ключа для строгого имени на вкладке **Signing** свойств проекта в Visual Studio. В качестве альтернативы вы можете использовать утилиту командной строки под названием **sn** (сокращение от **Strong Name**), которая умеет то, на что не способна Visual Studio, например добавлять подпись в сборку, которая была изначально построена с отсроченной подписью, или настраивать локально установленный .NET Framework, чтобы он игнорировал отсутствие действительной подписи для конкретных сборок с отсроченной подписью.

Уникальность строгого имени заключается в том, что в системах генерации ключей используются криптостойкие генераторы случайных чисел, и шансы того, что два человека создадут две пары ключей с одним и тем же

маркером открытого ключа, ничтожно малы. Уверенность в том, что сборка не была подделана, основывается на факте, что сборка со строгим именем подписана и только кто-то, владеющий закрытым ключом, способен был создать действительную подпись. Любая попытка изменить сборку после подписывания делает подпись недействительной.

Если закрытый ключ сборки становится публичным, любой может создать правдоподобно выглядящие сборки с соответствующим маркером ключа. Некоторые проекты с открытым исходным кодом преднамеренно публикуют оба ключа, чтобы каждый мог собирать компоненты из исходного кода. Это полностью исключает поддержку безопасности, которую может обеспечить маркер ключа, но это нормально, потому что Microsoft в настоящее время не рекомендует рассматривать строгие имена как функцию безопасности. Практика публикации вашего закрытого ключа с сильным именем подтверждает, что полезно иметь уникальное имя, даже без гарантии подлинности. .NET Core делает следующий шаг, позволяя компонентам иметь строгое имя, вообще не используя закрытый ключ. Вкупе с переходом Microsoft к разработке с открытым исходным кодом это означает, что теперь вы можете создавать и использовать собственные версии компонентов Microsoft, с таким же строгим именем, даже если Microsoft не публиковала свой закрытый ключ. Информацию о том, как работать с ключами, см. во врезке «Ключи, строгие имена и открытая подпись».

Microsoft использует тот же маркер на большинстве сборок в библиотеке классов .NET. (Компоненты .NET в Microsoft создаются разными группами, поэтому этот маркер является общим только для компонентов, являющихся частью .NET, но не для Microsoft в целом.) Вот полное название mscorelib, системной сборки, которая предлагает определения различных базовых типов, таких как `System.String`:

`mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089`

Кстати, это правильное название даже для самых последних версий .NET на момент написания. В нем записано 4.0.0.0, хотя .NET Framework теперь имеет версию 4.8, а .NET Core — 3.0. (В библиотеке .NET Core `mscorlib` не содержит ничего, кроме переадресации типов, в основном в `System.Private.CoreLib`, но номер версии совпадает.) Номера версий сборки имеют техническое значение, поэтому Microsoft не всегда обновляет номер версии в именах компонентов библиотеки согласно номерам маркетинговой версии — версии

не обязательно совпадают даже по основному номеру. Например, версия `mscorlib` для .NET 3.5 имела номер версии `2.0.0.0`.

Хотя маркер открытого ключа является необязательной частью имени сборки, версия — обязательна.

Версия

Все имена сборок содержат номер версии из четырех частей. Когда имя сборки представлено в виде строки (например, когда вы передаете его в качестве аргумента `Assembly.Load`), версия состоит из четырех десятичных целых чисел, разделенных точками (например, `4.0.0.0`). Двоичный формат, который IL использует для имен сборок и ссылок, ограничивает диапазон этих чисел — каждый фрагмент должен умещаться в 16-разрядное целое число без знака (`ushort`). Наивысшее допустимое значение в фрагменте версии по факту на единицу меньше максимального значения, что дает самый высокий допустимый номер версии `65534.65534.65534.65534`.

Каждая из четырех частей имеет собственное имя. Слева направо: основная версия, вспомогательная версия, сборка и ревизия. Тем не менее ни у одной части нет никакого особого значения. Некоторые разработчики используют определенные соглашения, но никто к ним не принуждает и никто не проверяет их исполнение. Общепринятое соглашение заключается в том, что любое изменение в публичном API требует изменения номера основной или вспомогательной версии, а изменение, которое может нарушить работу существующего кода, должно включать изменение основного номера. (Маркетинг — еще одна популярная причина значительных изменений версии.) Если обновление не предназначено для внесения каких-либо видимых изменений в поведение (кроме, возможно, исправления ошибки), достаточно изменить номер сборки. Номер ревизии можно использовать для различия двух компонентов, которые, по вашему мнению, были созданы на основе одного и того же исходного кода, но в разное время. Кроме того, некоторые люди связывают номера версий с ветвями в системе контроля версий, поэтому изменение только номера версии может указывать на исправление, примененное к версии, которая давно перестала получать основные обновления. Тем не менее вы можете придавать этим номерам собственный смысл. Что касается CLR, на самом деле есть только одна полезная вещь, которую вы можете проделать с номером версии: сравнить его с другим — они будут либо совпадать, либо один будет выше другого.

Номера версий в именах сборок библиотеки классов .NET игнорируют любые соглашения, которые я только что описал. Большинство компонентов имели одинаковый номер версии (`2.0.0.0`) в четырех основных обновлениях. В .NET 4.0 все они изменились на номер `4.0.0.0`, который на момент написания этой статьи все еще используется в последней версии .NET Framework (4.8). .NET Core 3.0 также использует 4 в качестве основной версии большинства компонентов библиотеки классов.

Вы обычно указываете номер версии, добавляя элемент `<Version>` внутрь `<PropertyGroup>` вашего `.csproj` файла. (В Visual Studio это можно сделать через пользовательский интерфейс: если вы откроете страницу `Properties` проекта, то на вкладке `Package` сможете настроить различные параметры, связанные с именованием. Поле `Package version` задает версию.) Система сборки использует это двумя способами: она устанавливает номер версии в сборке, но также, если вы генерируете пакет NuGet для своего проекта, по умолчанию присваивает этот же номер версии и пакету. Так как номера версий NuGet состоят из трех частей, обычно указывают только три, а для четвертой части версии сборки по умолчанию устанавливается ноль. (Если вы хотите указать все четыре цифры, обратитесь к документации, чтобы узнать, как установить версии сборки и NuGet по отдельности.)



Пакеты NuGet также имеют номера версий, и они не обязаны иметь какую-либо связь с версиями сборки. Многие авторы пакетов соглашаются делать их похожими, но это не является общепринятым. NuGet рассматривает компоненты номера версии пакета как имеющие особое значение: в нем приняты широко используемые правила семантического контроля версий. При этом используются версии с тремя частями: именованная основная, вспомогательная и заплатка.

Система сборки сообщает компилятору, какой номер версии использовать для имени сборки через атрибут уровня сборки. Более подробно я опишу атрибуты в главе 14, но конкретно этот довольно прост. Если он вам нужен, то система сборки обычно генерирует файл с именем `ProjectName.AssemblyInfo.cs` в подпапке папки `obj` вашего проекта. Он содержит различные атрибуты, описывающие детали сборки, включая атрибут `AssemblyVersion`, похожий на показанный в листинге 12.6.

Листинг 12.6. Указание версии сборки

```
[assembly: System.Reflection.AssemblyVersion("1.0.0.0")]
```

Компилятор C# обрабатывает его особым образом — он не применяет его вслепую, как это происходит с большинством атрибутов. Он анализирует номер версии и встраивает его так, как того требует формат метаданных .NET. Он также проверяет, соответствует ли строка ожидаемому формату и находятся ли числа в допустимом диапазоне.

Кстати, версия, которая является частью имени сборки, отличается от той, которая сохраняется с использованием стандартного механизма Win32 для встраивания версий. Большинство файлов .NET содержат оба варианта. По умолчанию система сборки будет использовать настройку `<Version>` для обоих параметров, но обычно файловая версия меняется чаще. Приведу пример. Хотя многие из файлов в текущей библиотеке классов .NET имеют номер версии с именем сборки `4.0.0.0`, если вы посмотрите информацию о версии файла в стиле Windows, вы обычно увидите что-то другое. Это было особенно важно с .NET Framework, в котором только один экземпляр любой основной версии может быть установлен одновременно — если на компьютере установлен .NET Framework 4.7.2, а вы устанавливаете .NET Framework 4.8, он заменяет версию 4.7.2. (.NET Core этого не делает — вы можете установить любое количество версий на одном компьютере.) Подобное обновление в сочетании с привычкой Microsoft сохранять одинаковые версии сборок в разных выпусках может усложнить понимание того, какая именно версия установлена, и здесь на помощь приходят файловые версии. На компьютере с установленным .NET Framework 4.0 sp1 его версия `mscorlib.dll` в Win32 имеет номер `4.0.30319.239`, но если вы установите .NET 4.8, она изменится на `4.8.4018.0`. (По мере выхода пакетов обновлений и других обновлений последняя часть продолжит расти.)

По умолчанию система сборки будет использовать `<Version>` как для сборки, так и для версии файла Windows, но если вы хотите установить версию файла отдельно, вы можете добавить `<FileVersion>` в файл вашего проекта. (Страница `Package` свойств проекта Visual Studio также позволяет вам его задать.) Внутри при этом используется другой атрибут, `AssemblyFileVersion`, который обрабатывается компилятором особым образом. В результате компилятор встраивает в файл ресурс с версией Win32, поэтому пользователи видят именно этот номер версии, когда щелкают правой кнопкой мыши по вашей сборке в проводнике Windows и выводят на экран свойства файла.

Этот файл с версией обычно является более подходящим местом для размещения номера версии, указывающего на происхождение сборки, нежели версия, которая входит в название сборки. Последний в действительности

является заявлением о поддерживаемой версии API, и любые обновления, разработанные для обеспечения полной обратной совместимости, с большой долей вероятности оставят его без изменений, а изменят только версию файла.

Номера версий и загрузка сборки

Поскольку номера версий являются частью имени сборки (и следовательно, ее определения), они в итоге являются и частью идентификатора типа. `System.String` в `mscorlib` версии `2.0.0.0` — это не то же самое, что тип с тем же именем в `mscorlib` версии `4.0.0.0`.

Обработка номеров версий сборки изменилась с появлением .NET Core. В .NET Framework, когда вы загружаете сборку со строгим именем по имени (либо неявно с помощью определяемых ей типов, либо явно с помощью `Assembly.Load`), CLR требует точного соответствия номер версии. .NET Core смягчает это требование и использует версию на диске, номер которой равен или больше запрашиваемого⁶. Это изменение обусловлено двумя факторами. Во-первых, экосистема разработки .NET стала полагаться на NuGet (который даже не существовал в течение большей части первого десятилетия существования .NET), что означает зависимость от достаточно большого числа внешних компонентов. Во-вторых, скорость изменений возросла — раньше нам зачастую приходилось годами ждать новые выпуски компонентов .NET. (Исправления безопасности и другие исправления ошибок могут появляться чаще, но новые функциональные возможности, как правило, появляются медленно и обычно большими блоками, как часть целой волны обновлений среди выполнения, библиотек и средств разработки.) Но сегодня редкий компонент может просуществовать месяц без каких-либо изменений в версии. Строгая политика управления версиями в .NET Framework теперь выглядит неконструктивной.

(Вообще, есть части системы сборки, предназначенные для изучения ваших зависимостей NuGet. Так можно выяснить конкретные версии каждого используемого вами компонента, чтобы потом автоматически генерировать файл конфигурации с огромным количеством правил подстановки версий, указывающие CLR использовать эти версии независимо от того, какую версию хочет использовать какая-либо отдельная сборка. Таким образом, даже

⁶ Можно настроить CLR для замены определенной отличной версии, но даже в этом случае загруженная сборка должна иметь точную версию, указанную в конфигурации.

если вы ориентируетесь на .NET Framework, система сборки по умолчанию по факту отключит строгое управление версиями.)

Другое изменение заключается в том, что .NET Framework учитывает версии сборок только в случае сборок со строгими именами. .NET Core проверяет, что номер версии сборки на диске равен или превышает требуемую версию независимо от того, имеет ли целевая сборка строгое имя.

Культура

До сих пор мы видели, что имена сборок включают в себя простое имя, номер версии и необязательно маркер открытого ключа. Но кроме этого, они содержат компонент культуры. (Культура представляет собой язык и набор соглашений, касающихся валюты, вариантов написания и форматы даты и т. д.) Он является обязательным, хотя наиболее часто ему оставляют значение по умолчанию: `neutral`, означающее, что сборка не содержит зависимого от культуры кода или данных. Что-то иное появляется только в сборках, которые содержат специфичные для культуры ресурсы. Региональные особенности имени сборки предназначены для поддержки локализации таких ресурсов, как изображения и строки. Чтобы показать, как именно это происходит, объясню механизм локализации, который ее использует.

Все сборки могут содержать встроенные двоичные потоки. (Конечно, вы можете поместить в эти потоки и текст — просто нужно выбрать подходящую кодировку.) Класс `Assembly` в API отражения предоставляет способ работы с ними напрямую, но чаще используется класс `ResourceManager` в пространстве имен `System.Resources`. Это гораздо удобнее, чем работа с необработанными двоичными потоками, поскольку `ResourceManager` определяет формат контейнера, который позволяет одному потоку содержать любое количество строк, изображений, звуковых файлов и других двоичных элементов, а `Visual Studio` имеет встроенный редактор для работы с этим форматом контейнера. Причина, по которой я упоминаю все это в середине раздела, который вроде бы касается имен сборок, заключается в том, что `ResourceManager` также обеспечивает поддержку локализации, а культура имени сборки является частью этого механизма. Чтобы показать, как это работает, приведу краткий пример.

Самым простым способом использования `ResourceManager` является добавление в ваш проект файла ресурсов в формате `.resx`. (Это не тот формат, который используется во время выполнения. Это формат XML, который

компилируется в требуемый `ResourceManager` двоичный формат. В большинстве систем управления версиями работать с текстом проще, чем с двоичным представлением.) Чтобы добавить один из них через диалоговое окно `Add New Item`, выберите категорию `Visual C# → General`, а затем выберите `Resources File`. Свой я назову `MyResources.resx`. Visual Studio отобразит собственный редактор ресурсов, который откроется в режиме редактирования строк, как показано на рис. 12.1. Как видите, я определил одну строку с именем `ColString` и значением `Color`.

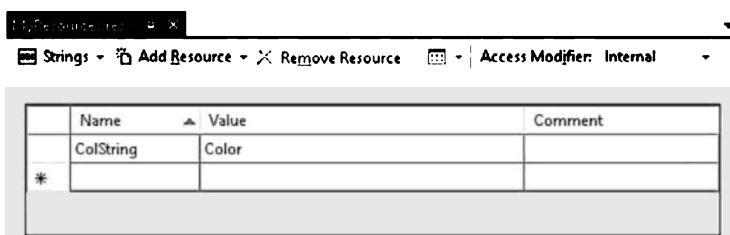


Рис. 12.1. Редактор файлов ресурсов в строковом режиме

Я могу получить это значение во время выполнения. Система сборки генерирует класс-обертку для каждого добавляемого вами файла `.resx` со статическим свойством для каждого ресурса, который вы определяете. Это очень упрощает поиск строкового ресурса, как показано в листинге 12.7.

Листинг 12.7. Получение ресурса с помощью класса-обертки

```
string colText = MyResources.ColString;
```

Класс-обертка скрывает детали, что обычно удобно, но в нашем случае подробности – это единственная причина, по которой я вообще показываю файл ресурсов, поэтому я показал, как непосредственно использовать `ResourceManager`, в листинге 12.8. Я включил весь исходный код файла, потому что здесь важны пространства имён – Visual Studio добавляет пространство имён вашего проекта по умолчанию перед именем потока встроенного ресурса, поэтому мне пришлось запрашивать `ResourceExample.MyResources` вместо просто `MyResources`. (Если бы я поместил ресурсы в папку в `Solution Explorer`, Visual Studio также включил бы имя этой папки в имя потока ресурсов.)

Пока что это всего лишь довольно многословный способ получить строку "Color". Однако теперь, когда у нас есть `ResourceManager`, я могу определить

некоторые локализованные ресурсы. Будучи британцем, я твердо знаю, как нужно правильно писать слово *color*. Но мне нужно адаптировать свою работу для своей преимущественно американской аудитории⁷. Но программа может работать лучше — она должна обеспечивать разные варианты написания для разных аудиторий. (Можно сделать следующий шаг и полностью изменить язык для стран, в которых какая-либо форма английского языка не является основным языком.) Фактически моя программа уже содержит весь код, необходимый для поддержки локализованного написания слова *color*. Мне просто нужно добавить альтернативный текст.

Листинг 12.8. Извлечение ресурса во время выполнения

```
using System;
using System.Resources;

namespace ResourceExample
{
    class Program
    {
        static void Main(string[] args)
        {
            var rm = new ResourceManager(
                "ResourceExample.MyResources", typeof(Program).Assembly);
            string colText = rm.GetString("ColString");
            Console.WriteLine("And now in " + colText);
        }
    }
}
```

Это можно сделать, добавив второй файл ресурсов с тщательно выбранным именем: *MyResources.en-GB.resx*. Оно похоже на оригинал, но с дополнительным фрагментом *.en-GB* перед расширением *.resx*. Это сокращение для «English-Great Britain», и это стандартизированное (хотя и не отражающее политическую ситуацию) название моей родной культуры. (Название культуры, обозначающее англоязычные части США, — *en-US*.) Добавив такой файл в свой проект, я могу добавить строковую запись с тем же именем, что и раньше, *ColString*, но на этот раз с правильным (в наших местах) значением, а именно *Colour*⁸. Если вы запустите приложение на компьютере, настроенном на использование британской локали, оно будет использовать

⁷ В американском английском корректным считается вариант «*color*», а в британском — «*colour*». — Примеч. ред.

⁸ Англия.

британское написание. Скорее всего, ваш компьютер настроен не так, поэтому, если хотите, можете добавить код в листинге 12.9 в самое начало метода `Main` в листинге 12.8, и это заставит .NET использовать британскую культуру при поиске ресурсов.

Листинг 12.9. Принудительное использование культуры

```
Thread.CurrentThread.CurrentCulture =
    new System.Globalization.CultureInfo("en-GB");
```

Как все это относится к сборкам? Что ж, если вы посмотрите на скомпилированный вывод, то увидите, что наряду с обычным исполняемым файлом и связанными файлами отладки Visual Studio создала подкаталог с именем `en-GB`, который содержит файл сборки с именем `ResourceExample.resources.dll`. (`ResourceExample` — это название моего проекта. Если вы создали проект под названием `SomethingElse`, вы увидите `SomethingElse.resources.dll`.) Имя этой сборки будет выглядеть следующим образом:

```
ResourceExample.resources, Version=1.0.0.0, Culture=en-GB,
PublicKeyToken=null
```

Номер версии и маркер открытого ключа будут совпадать с основным проектом — в моем примере я оставил номер версии по умолчанию и не давал своей сборке строгое имя. Но обратите внимание на `Culture`. Вместо обычного значения `neutral` на этом месте теперь `en-GB`, та же строка культуры, которую я указал в имени второго добавленного мной файла ресурсов. Если вы добавите больше файлов ресурсов с другими именами культур, вы получите папки, содержащие сборки для каждой культуры. Они называются сопутствующими ресурсными сборками.

Когда вы впервые запрашиваете ресурс `ResourceManager`, он будет искать сопутствующую ресурсную сборку с той же культурой, что и текущая культура пользовательского интерфейса потока. Поэтому он попытается загрузить сборку, используя имя, показанное пару абзацев назад. Если он ее не найдет, он попытается использовать более общее название культуры — если ему не удастся найти ресурсы `en-GB`, он будет искать культуру, называемую просто `en`, обозначающую английский язык без указания какого-либо конкретного региона. И только когда он не находит ни одной (или если находит соответствующие сборки, но они не содержат искомого ресурса), он возвращается к нейтральному ресурсу, встроенному в основную сборку.

Когда указана отличная от нейтральной культура, загрузчик сборки CLR будет искать ее в разных местах. Он попытается найти ее в подкаталоге с именем этой культуры. Вот почему Visual Studio поместил мою сопутствующую ресурсную сборку в папку `en-GB`.

Поиск специфичных для культуры ресурсов влечет за собой некоторые затраты времени выполнения. Они невелики, но если вы пишете приложение, которое никогда не будет локализовано, вы можете избежать затрат на функционал, который вы никогда не используете. Однако вы по-прежнему можете использовать `ResourceManager` — это более удобный способ встраивания ресурсов, чем непосредственное использование потоков ресурсов манифеста сборки. Чтобы избежать этих затрат, нужно сообщить .NET, что ресурсы, встроенные непосредственно в основную сборку, являются подходящими для конкретной культуры. Это можно сделать с помощью атрибута уровня сборки, показанного в листинге 12.10.

Листинг 12.10. Определение культуры для встроенных ресурсов

```
[assembly: NeutralResourcesLanguage("en-US")]
```

Когда приложение с этим атрибутом запускается на компьютере в обычной локали США, `ResourceManager` не будет пытаться искать ресурсы. Он просто будет довольствоваться встроенными в вашу основную сборку ресурсами.

Защита

В главе 3 я описал некоторые спецификаторы специальных возможностей, которые вы можете применять к типам и их элементам, например `private` или `public`. В главе 6 я показал некоторые дополнительные механизмы, доступные при использовании наследования. Стоит освежить в памяти этот функционал, потому что сборки в нем тоже играют определенную роль.

Кроме того, в главе 3 я представил ключевое слово `internal`, сообщив, что классы и методы с такой видимостью доступны только в одном и том же компоненте. Я выбрал слегка расплывчатый термин, потому что тогда я еще не добрался до сборок. Теперь, когда стало ясно, что такая сборка, я могу с уверенностью представить вам более точное описание ключевого слова `internal`. Его суть состоит в указании на то, что элемент или тип должен быть доступен только для кода в той же сборке. Аналогичным образом члены с видимостью `protected internal` доступны для кода в производных

типах, а также для кода, определенного в той же сборке. Аналогичный, но более строгий уровень защиты `protected private` делает члены доступными только для кода в производном типе, который определен в той же сборке⁹.

Итог

Сборка — это единица развертывания, почти всегда представляющая собой один файл, обычно с расширением .dll или .exe. Это контейнер для типов и кода. Тип принадлежит только к одной сборке, и эта сборка является частью определения типа — среды выполнения .NET способна различать два типа с одинаковыми именами в одном и том же пространстве имен, если они определены в разных сборках. Сборки имеют составное имя, состоящее из простого текстового имени, номера версии из четырех частей, строки культуры и (необязательно) маркера открытого ключа. Сборки с маркером открытого ключа называются сборками со строгим именем, которое является уникальным на глобальном уровне. Сборки могут быть развернуты вместе с приложением, которое их использует, или храниться в общесистемном хранилище. (В .NET Framework этот репозиторий представлял собой глобальный кэш сборок, и для его использования необходимо было давать сборкам строгое имя. .NET Core предоставляет общие копии встроенных сборок, и в зависимости от способа установки, он также может содержать общие копии платформ, таких как ASP.NET Core и WPF. При желании вы можете настроить отдельное хранилище пакетов времени выполнения, содержащее другие общие сборки, и избежать необходимости включать их в папки приложений.)

Среда выполнения по требованию может автоматически загружать сборки, что обычно происходит при первом запуске метода, содержащего код, который зависит от типа, определенного в соответствующей сборке. Вы также можете, если необходимо, загружать сборки явно.

Как я упоминал ранее, каждая сборка содержит исчерпывающие метаданные, описывающие содержащиеся в ней типы. В следующей главе я покажу, как получить доступ к этим метаданным во время выполнения.

⁹ Элементы с видимостью `internal` также доступны для дружественных сборок, т. е. для любых сборок, на которые ссылается атрибут `InternalsVisibleTo`, описываемый в главе 14.

Отражение

Среда CLR очень много знает о типах, которые определяют и используют наши программы. Она требует подробные метаданные от сборок, описывающие каждый член каждого типа, включая закрытые детали реализации. Эта информация нужна для работы важнейших функций, таких как JIT-компиляция и сборка мусора. Однако CLR не хранит эту информацию только для себя. API отражения предоставляет доступ к этой подробной информации о типах, поэтому ваш код может выяснить все, что доступно среде выполнения. Кроме того, вы можете с пользой применять отражение. Например, объект отражения, представляющий метод, не только описывает имя и сигнатуру метода, но также позволяет вызывать сам метод. Можно пойти еще дальше и генерировать код прямо во время выполнения.

Отражение особенно полезно в расширяемых средах, потому что они могут использовать его во время выполнения для адаптации своего поведения на основе структуры вашего кода. Например, панель **Properties** Visual Studio использует отражение, чтобы узнать, какие публичные свойства предлагает компонент. Поэтому, если вы пишете компонент, который может отображаться в конструкторе дизайна, например элемент пользовательского интерфейса, вам не нужно ничего предпринимать, чтобы сделать его свойства доступными для редактирования, — Visual Studio найдет их автоматически.



Многие основанные на отражении платформы, которые могут автоматически находить то, что им нужно знать, дополнительно позволяют компонентам явно дополнять эту информацию. Например, несмотря на то что вам не нужно делать ничего особенного в панели **Properties** для поддержки редактирования, при желании можно настроить механизмы категоризации, описания и редактирования. Обычно это достигается с помощью атрибутов, которые являются темой главы 14.

Типы отражения

API отражения определяет различные классы в пространстве имен `System.Reflection`. Эти классы имеют структурные отношения, которые отражают принцип работы сборок и системы типов. Например, содержащая сборка типа является частью его определения, поэтому класс отражения, представляющий тип (`Type`), имеет свойство `Assembly`, которое возвращает содержащий его объект `Assembly`¹. И вы можете перемещаться по этим отношениям в обоих направлениях — вы можете получить все типы в сборке из свойства `DefinedTypes` класса `Assembly`. Приложение, расширяемое путем загрузки DLL загружаемых модулей, обычно использует это для поиска типов, которые предоставляет каждый модуль. На рис. 13.1 показаны типы отражений, соответствующие типам .NET, их членам и компонентам, которые их содержат. Стрелки отражают отношения включения. (Как и в случае сборок и типов, по ним можно переходить в обоих направлениях.)

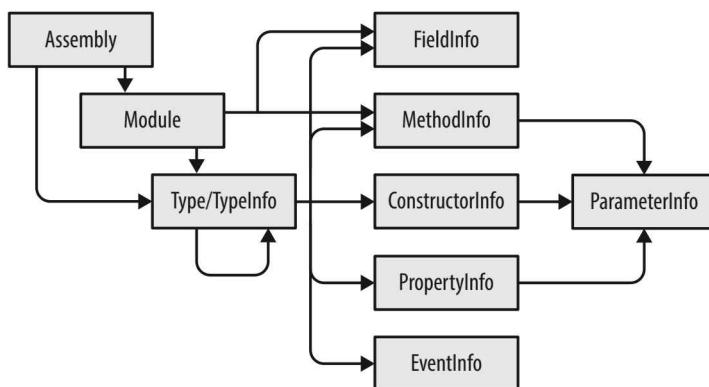


Рис. 13.1. Иерархия отражения

На рис. 13.2 показана иерархия наследования для этих типов. На нем показана пара дополнительных абстрактных типов, `MemberInfo` и `MethodBase`, которые совместно используются различными классами отражения, имеющими много общего. Например, у конструкторов и методов есть списки параметров, и механизм их проверки обеспечивается общим базовым классом,

¹ По историческим причинам, о которых я расскажу позже, часть этой функциональности находится в производном типе, называемом `TypeInfo`. Но чаще всего вы будете сталкиваться с базовым классом `Type`.

MethodBase. Все члены типов имеют определенные общие характеристики, такие как видимость, поэтому все, что является (или может быть) членом типа, представляется в отражении объектом, который является производным от `MethodInfo`.

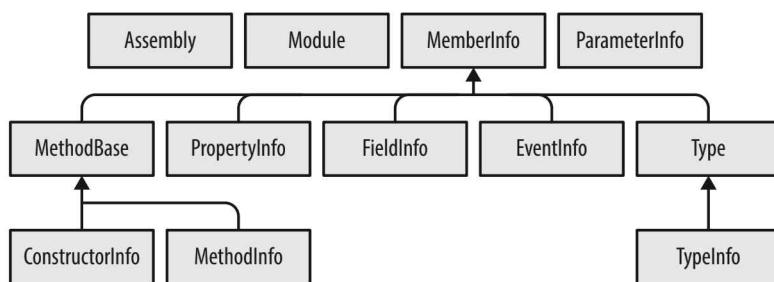


Рис. 13.2. Отражение иерархии наследования

Assembly

Класс `Assembly` представляет, что вполне предсказуемо, одну сборку. Если вы пишете систему модулей или какой-то другой тип библиотеки, который должен загружать и использовать предоставленные пользователем DLL (например, исполнитель юнит-тестов), тип `Assembly` будет вашей отправной точкой. Как было показано в главе 12, статический метод `Assembly.Load` принимает имя сборки и возвращает объект для этой сборки. (Этот метод при необходимости загрузит сборку, но если она уже загружена, он просто вернет ссылку на соответствующий объект `Assembly`.) Но есть и другие способы добраться до объектов такого рода.

Класс `Assembly` определяет три контекстно-зависимых статических метода, каждый из которых возвращает `Assembly`. Метод `GetEntryAssembly` возвращает объект, представляющий файл EXE, содержащий метод `Main` вашей программы. Метод `GetExecutingAssembly` возвращает сборку, которая содержит метод, из которого вы его вызвали. `GetCallingAssembly` поднимается по стеку на один уровень и возвращает сборку, содержащую код, вызвавший метод, который вызвал `GetCallingAssembly`.

`GetCallingAssembly` иногда может быть полезен при ведении журнала диагностики, поскольку предоставляет информацию о коде, вызвавшем ваш метод. Метод `GetExecutingAssembly` не так полезен: вы, вероятно, уже знаете, в какой сборке будет находиться код, поскольку вы его разработчик. Воз-

можно, вам все же потребуется объект `Assembly` для компонента, который вы пишете, но есть и другие способы. Объект `Type`, описанный в следующем разделе, содержит свойство `Assembly`. Листинг 13.1 использует его, чтобы получить `Assembly` через содержащий класс. Опыт показывает, что это работает быстрее, что неудивительно, потому что выполняется меньше работы — в обеих техниках извлекаются объекты отражения, но в одной из них также должен проверяться стек.

Листинг 13.1. Получение собственной сборки через `Type`

```
class Program
{
    static void Main(string[] args)
    {
        Assembly me = typeof(Program).Assembly;
        Console.WriteLine(me.FullName);
    }
}
```



Оптимизация JIT-компилятора может иногда давать неожиданные результаты в случае с `GetExecutingAssembly` и `GetCallingAssembly`. Оптимизация встраивания методов и завершающих вызовов может заставить эти методы возвращать сборку для методов, которые находятся на один кадр стека дальше, чем ожидалось. Вы можете предотвратить оптимизацию встраивания, аннотировав метод с помощью `MethodImplAttribute`, передав флаг `NoInlining` из перечисления `MethodImplOptions`. (Атрибуты описаны в главе 14.) Нет способа явно отключить оптимизацию завершающих вызовов, но они применяются только когда конкретный вызов метода — это последнее, что метод делает перед возвратом.

Если нужно использовать сборку из определенного места на диске, вы можете использовать метод `LoadFile`, описанный в главе 12. В качестве альтернативы имеется другой статический метод класса `Assembly`, `ReflectionOnlyLoadFrom`. Он загружает сборку таким образом, что вы можете получить информацию о ее типе, но никакой код в сборке не будет выполняться и никакие сборки, от которых она зависит, не будут загружены автоматически. Это подходящий способ загрузки сборки, если вы пишете инструмент, который отображает или иным образом обрабатывает информацию о компоненте, но не хочет выполнять его код. Есть несколько причин, по которым с помощью такого инструмента важно не загружать сборку обычным способом. Загрузка сборки и проверка ее типов могут иногда запускать выполнение

кода в этой сборке (например, статических конструкторов). Кроме того, если вы загружаете ее только для целей отражения, архитектура процессора не имеет существенного значения, поэтому вы можете загрузить 32-разрядную DLL-библиотеку в 64-разрядный процесс или проверить сборку только для ARM в процессе x86.

Получив `Assembly` посредством любого из вышеупомянутых механизмов, вы можете много чего о ней узнать. Например, свойство `FullName` предоставляет отображаемое имя. Или вы можете вызвать `GetName`, который возвращает объект `AssemblyName`, давая легкий программный доступ ко всем компонентам имени сборки.

Вы можете получить список всех других сборок, от которых зависит конкретная сборка, вызвав `GetReferencedAssemblies`. Если вы вызовете этот метод для написанной вами сборки, она не обязательно вернет все сборки, которые вы видите в узле `Dependencies` в обозревателе решений Visual Studio, потому что компилятор C# удаляет неиспользуемые ссылки.

Сборки содержат типы, поэтому вы можете увидеть объекты `Type`, представляющие эти типы, вызвав метод `GetType` объекта `Assembly` и передав ему имя требуемого типа, включая его пространство имен. Он вернет `null`, если тип не найден, если только вы не вызовете одну из перегрузок, которые дополнительны принимают `bool`. Передав в них `true`, вы получите исключение, если тип не найден. Существует также перегрузка, которая принимает два аргумента `bool`, второй из которых позволяет передать значение `true`, чтобы запросить поиск без учета регистра. Все эти методы будут возвращать типы `public` или `internal`. Вы также можете запросить вложенный тип, указав имя содержащего типа, добавив символ +, а затем имя вложенного типа. Листинг 13.2 получает объект `Type` для типа с именем `Inside`, вложенный в тип с именем `ContainedType` в пространстве имен `MyLib`. Это работает, даже если вложенный тип является закрытым.

Листинг 13.2. Получение вложенного типа из сборки

```
Type nt = someAssembly.GetType("MyLib.ContainingType+Inside");
```

Класс `Assembly` также предоставляет свойство `DefinedTypes`, которое возвращает коллекцию, содержащую объект `TypeInfo` для каждого определяемого сборкой типа (верхнего уровня или вложенного), а также `ExportedTypes`, которое возвращает только открытые типы, кроме того, что возвращает объекты `Type`, а не полные объекты `TypeInfo`. (Различие между `TypeInfo`

и `Type` описано в разделе «`Type` и `TypeInfo`» на с. 712.) В их число также войдут вложенные типы с видимостью `public`. Он не будет включать типы `protected`, вложенные в типы `public`, что может удивлять, поскольку такие типы доступны снаружи сборки (хотя только для классов, производных от содержащего типа).

Помимо возвращаемых типов `Assembly` также умеет создавать их экземпляры с помощью метода `CreateInstance`. Передав только полное имя типа в виде строки, вы получите его экземпляр в том случае, если тип является открытым и имеет конструктор без аргументов. Существует перегрузка, которая позволяет вам работать с типами с отличной от `public` видимостью и типами с конструкторами, которые требуют аргументов. Однако его использование является более сложным, поскольку он также принимает аргументы, указывающие, требуется ли учет регистра для имени типа, а также объект `CultureInfo`, определяющий правила, используемые для сравнений без учета регистра, — разные страны имеют разное представление о том, как работают такие сравнения. Также у него есть аргументы для управления более сложными сценариями. Конечно, в большинство из них можете передать `null`, как показано в листинге 13.3.

Листинг 13.3. Динамическое конструирование

```
object o = asm.CreateInstance(
    "MyApp.WithConstructor",
    false,
    BindingFlags.Public | BindingFlags.Instance,
    null,
    new object[] { "Constructor argument" },
    null,
    null);
```

Пример создает экземпляр типа `WithConstructor` в пространстве имен `MyApp` в сборке, на которую ссылается `asm`. Аргумент `false` указывает на то, что нам требуется точное совпадение имени с учетом регистра. `BindingFlags` указывает на то, что мы ищем публичный конструктор экземпляра. (См. врезку «`BindingFlags`».) Первый аргумент `null` — это место, куда вы можете передать объект `Binder`, позволяющий настроить поведение, когда предоставленные вами аргументы не точно соответствуют типам требуемых аргументов. Опуская его, я указываю, что предоставленные мной аргументы будут соответствовать точно. (Иначе я получу исключение.) Аргумент `object[]` содержит список аргументов, которые я хотел бы передать конструктору, — в данном случае это единственная строка. Предпоследний `null` — это то

место, куда бы отправилась культура, если бы использовались сравнения без учета регистра или автоматические преобразования между числовыми типами и строками. Но так как я не делаю ни того ни другого, то могу это опустить. Последний аргумент когда-то поддерживал сценарии, которые уже устарели, поэтому он всегда должен содержать `null`.

BINDINGFLAGS

Многие из API отражения принимают аргумент типа перечисления `BindingFlags`, чтобы определить, какие члены следует возвращать. Например, вы можете установить `BindingFlags.Public`, чтобы указать, что вам нужны только открытые члены или типы или `BindingFlags.NonPublic` для элементов, которые не являются открытыми, а можно и объединить оба флага, чтобы указать, что вы хотите получить и то и другое.

Имейте в виду, что есть комбинации, которые не вернут вообще ничего. Например, при работе с членами необходимо включить либо `BindingFlags.Instance`, `BindingFlags.Static`, либо оба, потому что все члены типа являются либо тем, либо другим (аналогично `BindingFlags.Public` и `BindingFlags.NonPublic`).

Часто методы, которые принимают `BindingFlags`, предлагают перегрузку, которая этого не делает. Обычно по умолчанию при этом запрашиваются открытые члены, как экземпляры, так и статические (т. е. `BindingFlags.Public | BindingFlags.Static | BindingFlags.Instance`).

`BindingFlags` определяет множество параметров, но не все из них применимы в каждом сценарии. Например, есть значение `FlattenHierarchy`; оно используется для API отражения, которые возвращают члены типа: если этот флаг установлен, то наряду с элементами, определенными указанным классом, будут рассматриваться элементы, определенные базовым. Этот параметр не применим к `Assembly.CreateInstance`, поскольку нельзя напрямую использовать конструктор базового класса для создания производного типа.

Если сборка содержит несколько файлов, вы можете получить полный их список с помощью метода `GetFiles`, который возвращает массив объектов `FileStream`, тип, который .NET использует для представления файлов. Если вы передадите `true`, список будет включать любые потоки ресурсов, хранящиеся в виде отдельных файлов, внешних по отношению к основной сборке. В противном случае будет предоставлен один поток на модуль. В качестве альтернативы можно вызвать `GetModules`, который также возвращает массив, содержащий составляющие сборку модули, но вместо возврата объектов `FileStream` он возвращает объекты `Module`.

Module

Класс `Module` представляет собой один из модулей, составляющих сборку. Большинство сборок являются одномодульными (и в .NET Core они такие всегда), поэтому вам не придется часто использовать этот тип. Он нужен, когда вы генерируете код во время выполнения, потому что вам потребуется указать .NET, в каком модуле размещать генерируемый код, поэтому даже в обычных одномодульных сценариях вы должны четко указывать такой модуль. Но если вы не генерируете новые компоненты во время выполнения, зачастую можно полностью игнорировать класс `Module` — как правило, всего, что вам нужно, можно добиться с помощью других типов из API отражения. (API .NET для генерации кода во время выполнения выходят за рамки этой книги.)

Если вам по какой-то причине нужен объект `Module`, вы можете извлечь модули из свойства `Modules` содержащего их объекта `Assembly`. В качестве альтернативы можно использовать любой из типов API, описанных в следующих разделах и являющихся производными от `MethodInfo`. (Такие типы показаны на рис. 13.2.) Такой тип содержит свойство `Module`, возвращающее модуль, в котором определен соответствующий член.

Класс `Module` содержит свойство `Assembly`, которое возвращает ссылку на содержащую модуль сборку. Свойство `Name` возвращает имя файла для этого модуля, а `FullyQualifiedName` предоставляет имя файла, включая полный путь.

Подобно классу `Assembly`, `Module` определяет метод `GetType`. В сборке с одним модулем это будет неотличимо от того же метода в классе `Assembly`, но если вы разбили код своей сборки на несколько модулей, эти методы будут предоставлять доступ только к типам, определенным в модуле, на который у вас имеется ссылка.

Как ни странно, класс `Module` также определяет свойства `GetField`, `GetFields`, `GetMethod` и `GetMethods`. Они предоставляют доступ к методам и полям в глобальной области видимости. Вы никогда не увидите их в C#, потому что сам язык требует, чтобы все поля и методы были определены внутри типа, но CLR допускает глобальные области видимости методов и полей, так что API отражения должен иметь возможность их выдавать. (C++/CLI способен создавать глобальные поля.)

MemberInfo

Как и все классы, которые я описываю в этом разделе, `MemberInfo` является абстрактным. Однако, в отличие от остальных, он не относится к какой-то одной особенности системы типов. Это общий базовый класс, предоставляющий общие функциональные возможности всем типам, представляющим элементы, которые могут быть членами других типов. Так что это базовый класс `ConstructorInfo`, `MethodInfo`, `FieldInfo`, `PropertyInfo`, `EventInfo` и `Type`, потому что все они могут быть членами других типов.

На самом деле в C# все типы, кроме `Type`, должны быть членами какого-то другого типа (хотя, как вы только что видели в предыдущем разделе, некоторые языки допускают область видимости методов и полей в рамках модуля, а не типа).

`MemberInfo` определяет общие свойства, необходимые для всех членов типа. Конечно, у него есть свойство `Name`, а также `DeclaringType`, который ссылается на объект `Type` для содержащего типа элемента; оно возвращает `null` для невложенных типов, а также методов и полей в области видимости модуля. `MemberInfo` также определяет свойство `Module`, которое ссылается на содержащий модуль, независимо от того, находится ли рассматриваемый элемент в области видимости модуля или является членом типа.

Подобно `DeclaringType`, `MemberInfo` определяет `ReflectedType`, указывающий на тип, из которого был получен `MemberInfo`. Они часто будут одинаковыми, но могут отличаться при наследовании. Листинг 13.4 показывает это различие.

Листинг 13.4. Сравнение `DeclaringType` и `ReflectedType`

```
class Base
{
    public void Foo()
    {
    }
}

class Derived : Base
{
}

class Program
{
```

```

static void Main(string[] args)
{
    MethodInfo bf = typeof(Base).GetMethod("Foo");
    MethodInfo df = typeof(Derived).GetMethod("Foo");

    Console.WriteLine("Base Declaring: {0}, Reflected: {1}",
                      bf.DeclaringType, bf.ReflectedType);
    Console.WriteLine("Derived Declaring: {0}, Reflected: {1}",
                      df.DeclaringType, df.ReflectedType);
}
}

```

Пример получает `MethodInfo` для методов `Base.Foo` и `Derived.Foo`. (`MethodInfo` наследуется от `MemberInfo`.) Это просто разные способы описания одного и того же метода — `Derived` не определяет собственный `Foo`, а просто наследует определенный в `Base`. Программа производит следующий вывод:

```

Base Declaring: Base, Reflected: Base
Derived Declaring: Base, Reflected: Derived

```

При извлечении информации о `Foo` через объект `Type` класса `Base` как `DeclaringType`, так и `ReflectedType`, что неудивительно, относятся к `Base`. Однако когда мы получаем информацию о методе `Foo` через тип `Derived`, `DeclaringType` сообщает нам, что метод определен в `Base`, а `ReflectedType` — что мы получили этот метод через тип `Derived`.



Поскольку `MemberInfo` запоминает, из какого типа вы его получили, сравнение двух объектов `MemberInfo` не является надежным способом определить, ссылаются ли они на одно и то же. Сравнение `bf` и `df` в листинге 13.4 с помощью оператора `==` или их метода `Equals` вернет `false`, несмотря на тот факт, что обе переменные ссылаются на `Base.Foo`. В каком-то смысле это логично — это разные объекты, и их свойства не полностью идентичны, поэтому очевидно, что они не равны. Но если вы не знаете о свойстве `ReflectedType`, такое поведение может стать неожиданностью.

Удивляет, что `MemberInfo` не предоставляет никакой информации о видимости описываемого члена. Это может показаться странным, поскольку в C# все конструкции, соответствующие производным от `MemberInfo` типам (например, конструкторы, методы или свойства), могут иметь префикс `public`, `private` и т. д. API отражения дает доступ к этой информации, но не через базовый класс `MemberInfo`. Это связано с тем, что CLR обрабатывает

видимость для определенных типов элементов немного иначе, чем C#. С точки зрения CLR свойства и события не имеют собственной видимости. Вместо этого их видимость контролируется на уровне отдельных методов. Это позволяет свойствам `get` и `set` иметь разные уровни видимости, это же относится к методам доступа к событию. Конечно, если нужно, то в C# мы можем управлять видимостью свойств независимо. C# вводит нас в заблуждение тем, что позволяет указывать единый уровень видимости для всего свойства. Но это просто сокращенная запись постановки обоих методов доступа на один и тот же уровень. Загвоздка в том, что это позволяет нам указать видимость для свойства, а затем отличную видимость для одного из членов, как показано в листинге 13.5.

Листинг 13.5. Видимость метода доступа к свойству

```
public int Count
{
    get;
    private set;
}
```

Это может ввести в заблуждение, потому что, несмотря на то как все это выглядит, видимость `public` не распространяется на все свойство. Эта видимость на уровне свойства просто сообщает компилятору, что использовать в случае методов доступа, которые не указывают собственный уровень видимости. Первая версия C# требовала, чтобы оба средства доступа к свойствам имели одинаковую доступность, поэтому можно было указывать его для всего свойства. (Все еще существует аналогичное ограничение для событий.) Но это было необоснованное ограничение — CLR всегда позволяла каждому методу доступа иметь различную видимость. Теперь C# тоже это поддерживает, но по историческим причинам синтаксис для этого смущает своей несимметричностью. С точки зрения CLR в листинге 13.5 просто оказывается, что `get` следует сделать публичным, а `set` — закрытым. Листинг 13.6 лучше показывает, что на самом деле происходит.

Но такая запись недопустима, потому что C# требует указывать доступность более видимого метода доступа на уровне свойства. Это упрощает синтаксис, когда оба свойства имеют одинаковую видимость, но выглядит странно, когда она разная. Более того, синтаксис в листинге 13.5 (т. е. синтаксис, который фактически поддерживается компилятором) заставляет все выглядеть так, как будто нам следует указывать видимость в трех местах:

в свойстве и обоих методах доступа. CLR такое не поддерживает, поэтому компилятор выдаст ошибку, если вы попытаетесь указать видимость обоих методов доступа к свойству или событию. Таким образом, это не видимость самих свойства или события. (Представьте, если бы это было так — что бы вообще значил тот факт, что свойство имеет видимость `public`, его метод `get` — `internal`, а `set` — `private`?) Следовательно, не все, что происходит от `MemberInfo`, имеет отдельную видимость, поэтому API отражения содержит свойства, представляющие видимость ниже в иерархии классов.

Листинг 13.6. Как CLR видит доступность свойств

```
// Не будет компилироваться, хотя, по идее, должно
int Count
{
    public get;
    private set;
}
```

Type и TypeInfo

Класс `Type` представляет определенный тип. Он используется более широко, чем любой другой класс из этой главы, поэтому он единственный располагается в пространстве имен `System`, тогда как остальные — в `System.Reflection`. Его проще всего получить, потому что в C# есть оператор, предназначенный именно для этой задачи: `typeof`. Я уже включал его в несколько примеров, но листинг 13.7 демонстрирует его отдельно. Как видите, можно использовать либо встроенное имя, например `string`, либо имя обычного типа, например `IDisposable`. Вы также можете добавить пространство имен, но в этом нет необходимости, когда пространство имен типа находится в области видимости.

Листинг 13.7. Получение Type с помощью typeof

```
Type stringType = typeof(string);
Type disposableType = typeof(IDisposable);
```

Кроме того, как я упоминал в главе 6, тип `System.Object` (или `object`, как мы обычно записываем его в C#) предоставляет метод экземпляра `GetType`, который не принимает аргументов. Его можно вызвать для любой переменной ссылочного типа, чтобы получить тип объекта, на который она ссылается. Это не обязательно будет тот же тип, что и сама переменная, поскольку переменная способна ссылаться на экземпляр производного типа. Вы также

можете вызвать этот метод для любой переменной значимого типа, и поскольку значимые типы не поддерживают наследование, он всегда будет возвращать объект типа для статического типа переменной.

Поэтому, имея объект, значение или идентификатор типа (например, `string`), получить объект `Type` очень просто. Есть множество других мест, откуда можно получить объекты `Type`.

В дополнение к `Type` у нас также есть `TypeInfo`. Он появился в ранних версиях .NET Core для того, чтобы `Type` мог служить упрощенным идентификатором, а `TypeInfo` использоваться в качестве механизма отражения типа. Это было отклонением от обычного принципа работы `Type` в .NET Framework, где он играет обе роли. Некоторые считают, что эта двойная роль была ошибкой, потому что, если вам нужен только идентификатор, `Type` излишне тяжеловесный вариант. Первоначально предполагалось, что платформа .NET Core будет существовать отдельно от .NET Framework, не требуя строгой совместимости, поэтому она, как тогда казалось, давала возможность решить старые проблемы проектирования. Однако после того, как Microsoft приняла решение о том, что .NET Core станет основой всех будущих версий .NET, появилась необходимость привести `Type` в соответствие с тем, как он всегда работал с .NET Framework. Однако к этому времени в .NET Framework уже появился `TypeInfo`. И, чтобы минимизировать несовместимость с .NET Core 1, именно к нему на некоторое время на уровне типов были добавлены новые функции отражения. .NET Core 2.0 был приведен в соответствие с .NET Framework, но это означало, что разделение функциональности между `Type` и `TypeInfo` теперь стало просто результатом этих поправленных добавлений. `TypeInfo` содержит члены, добавленные в течение короткого периода между его появлением и решением вернуться к старому методу. В тех случаях, когда у вас имеется `Type`, но нужно использовать функцию, специфичную для `TypeInfo`, вы можете получить его из типа, вызвав `GetTypeInfo`.

Как вы уже видели, вы можете получать объекты `Type` из сборки либо по имени, либо в составе полного списка. Типы отражений, производные от `MemberInfo`, также предоставляют ссылку на содержащий их тип через `DeclaringType`. (`Type` является производным от `MemberInfo`, поэтому также содержит это свойство, полезное при работе с вложенными типами.)

Вы также можете вызвать собственный статический метод `GetType` класса `Type`. Если передать строку только с указанием пространства имен, он будет искать именованный тип в системной сборке с именем `mscorlib`, а также

в сборке, из которой вы вызывали этот метод. Однако вы можете передать и имя с указанием сборки, включающее в себя имя сборки и имя типа. Имя такого вида начинается с имени типа, включающего пространство имен, за которым следуют запятая и имя сборки. Например, вот имя с указанием сборки класса `System.String` в .NET 4.8 (разделенное на две строки, чтобы поместиться на странице):

```
System.String, mscorelib, Version=4.0.0.0, Culture=neutral,  
PublicKeyToken=b77a5c561934e089
```

Узнать имя, соответствующее типу сборки, можно через свойство `Type.AssemblyQualifiedName`. Имейте в виду, что оно не всегда будет совпадать с тем, что было запрошено. При работе в .NET Core, если вы передадите предыдущее имя типа в `Type.GetType`, это сработает, но если запросить у возвращенного `Type` его `AssemblyQualifiedName`, то вы увидите вот что:

```
System.String, System.Private.CoreLib, Version=4.0.0.0,  
Culture=neutral, PublicKeyToken=7cec85d7bea7798e
```

Единственная причина, по которой метод работает, если вы передаете либо первую строку, либо просто `System.String`, заключается в том, что `mscorelib` все еще присутствует для обратной совместимости. Я говорил об этом в предыдущей главе, но стоит кратко повторить. В .NET Framework сборка `mscorelib` содержит основные типы библиотеки классов, но в .NET Core этот код был перемещен в другое место. `mscorelib` все еще существует, но содержит только записи переадресации типов, указывающие, в какой сборке теперь находится каждый класс. Например, он переадресует `System.String` в его новый дом, которым на момент написания являлась сборка `System.Private.CoreLib`.

Существует соответствующий метод `ReflectionOnlyGetType`, который работает аналогичным образом, но загружает сборки только в контексте отражения, точно так же, как метод `ReflectionOnlyLoadFrom` класса `Assembly`, описанный ранее.

Наряду со стандартными свойствами `MethodInfo`, такими как `Module` и `Name`, классы `Type` и `TypeInfo` добавляют различные собственные свойства. Унаследованное свойство `Name` содержит имя без дополнительного определения, поэтому `Type` добавляет свойство `Namespace`. Все типы ограничены областью видимости сборки, поэтому `Type` определяет свойство `Assembly`. (Конечно, вы можете попасть туда через `Module.Assembly`, но удобнее использовать свойство `Assembly`.) Он также определяет свойство `BaseType`, хотя для некоторых

типов оно будет содержать `null` (например, ненаследованные интерфейсы и объект типа для класса `System.Object`).

Поскольку `Type` может представлять все виды типов, есть свойства, которые можно использовать для уточнения: `IsArray`, `IsClass`, `IsEnum`, `IsInterface`, `IsPointer` и `IsValueType`. (В сценариях взаимодействия также можно получить объекты `Type` для типов, отличных от .NET, поэтому существует свойство `IsCOMObject`.) Если `Type` представляет класс, то можно использовать ряд свойств, сообщающих, какой именно класс вы получили: `IsAbstract`, `IsSealed` и `IsNested`. Последнее применимо как к значимым типам, так и к классам.

`Type` также определяет многочисленные свойства, предоставляющие информацию об области видимости типа. Для невложенного типа `IsPublic` сообщает, является ли он `public` или `internal`, но для вложенных типов все сложнее. `IsNestedAssembly` указывает на вложенный тип `internal`, в то время как `IsNestedPublic` и `IsNestedPrivate` — на вложенные типы `public` и `private`. Вместо обычной терминологии семейства C (`protected`) в CLR используется термин *family*, поэтому методы теперь называются так: `IsNestedFamily` для `protected`, `IsNestedFamORAssem` для `protected internal` и `IsNestedFamANDAssem` для `protected private`.

Класс `TypeInfo`, в свою очередь, предоставляет методы для обнаружения связанных объектов отражения. (Все свойства этого абзаца определены в `TypeInfo`, а не в `Type`. Как обсуждалось ранее, это произошло по воле случая и обусловлено временем появления.) Большинство из них представлены в двух формах: одна для случая, когда вы знаете название того, что ищете, а другая — когда вы хотите получить полный список всех элементов указанного вида. Например, у нас имеются `DeclaredConstructors`, `DeclaredEvents`, `DeclaredFields`, `DeclaredMethods`, `DeclaredNestedTypes` и `DeclaredProperties`, а также их аналоги: `GetDeclaredConstructor`, `GetDeclaredEvent`, `GetDeclaredField`, `GetDeclaredMethod`, `GetDeclaredNer` и `GetredeperTested`.

Класс `Type` позволяет обнаруживать отношения совместимости типов. Вызвав метод `IsSubclassOf` типа, вы можете узнать, является ли он производным от другого. Наследование — не единственная причина, по которой один тип может быть совместим со ссылкой другого типа. Переменная, тип которой является интерфейсом, способна ссылаться на экземпляр любого типа, который реализует этот интерфейс, независимо от его базового класса. Поэтому класс `Type` содержит более общий метод с именем `IsAssignableFrom`, который используется в листинге 13.8.

Листинг 13.8. Проверка совместимости типов

```
Type stringType = typeof(string);
Type objectType = typeof(object);
Console.WriteLine(stringType.IsAssignableFrom(objectType));
Console.WriteLine(objectType.IsAssignableFrom(stringType));
```

Результатом будет `False`, а затем `True`, потому что вы не можете взять ссылку на экземпляр `object` и присвоить ее переменной типа `string`, но можно взять ссылку на экземпляр `string` и присвоить ее переменной типа `object`.

Наряду с информацией о типе и его отношениях с другими типами класс `Type` дает возможность использовать члены типа во время выполнения. Он определяет метод `InvokeMember`, точное значение которого зависит от того, какой тип элемента вы вызываете, — это может означать вызов метода или, например, получение или установку свойства или поля. Поскольку некоторые типы элементов поддерживают несколько видов вызовов (например, `get` и `set`), вам необходимо указать, какую конкретную операцию вы запрашиваете. Листинг 13.9 использует `InvokeMember` для вызова метода, идентифицируемого по его имени (строковый аргумент) в экземпляре типа, также идентифицируемого по имени, который он создает динамически. Это показывает, как отражение может использоваться для работы с типами и членами, идентификационные данные которых неизвестны до времени выполнения.

Листинг 13.9. Вызов метода с помощью `InvokeMember`

```
public static object CreateAndInvokeMethod(
    string typeName, string member, params object[] args)
{
    Type t = Type.GetType(typeName);
    object instance = Activator.CreateInstance(t);
    return t.InvokeMember(
        member,
        BindingFlags.Instance | BindingFlags.Public |
        BindingFlags.InvokeMethod,
        null,
        instance,
        args);
}
```

В этом примере сначала создается экземпляр указанного типа. При этом используется немного иной подход к динамическому созданию, чем тот, который я показал ранее и который использует `Assembly.CreateInstance`.

Здесь я вызываю `Type.GetType` для поиска типа, а затем использую класс, который до этого не упоминал, а именно `Activator`. Работа этого класса заключается в создании экземпляров объектов, тип которых вы определили во время выполнения. Его функциональные возможности частично совпадают с `Assembly.CreateInstance`, но в нашем случае это наиболее удобный способ перехода от `Type` к новому экземпляру этого типа. После этого я использовал `InvokeMember` объекта `Type` для вызова указанного метода. Как и в листинге 13.3, мне пришлось указать флаги привязки, чтобы указать, какой тип элемента я ищу, а также что с ним делать, — в данном случае я пытаюсь вызвать метод (в отличие, например, от установки значения свойства). Аргумент `null`, как и в листинге 13.3, обозначает то место, где я бы указал `Binder`, если бы требовалось поддерживать автоматическое приведение типов аргументов метода.

Обобщенные типы

Поддержка .NET обобщений усложняет роль класса `Type`. Наряду с представлением обычного необобщенного типа `Type` способен представлять не только конкретный экземпляр обобщенного типа (например, `List<int>`), но также и несвязанный обобщенный тип (например, `List<>`, хотя это недопустимый идентификатор типа во всех случаях, кроме одного крайне специфического сценария). Листинг 13.10 показывает, как получить оба вида объектов `Type`.

Листинг 13.10. Объекты `Type` для обобщенных типов

```
Type bound = typeof(List<int>);  
Type unbound = typeof(List<>);
```

Оператор `typeof` — это единственное место, где в C# вы можете использовать несвязанный идентификатор обобщенного типа. Во всех других контекстах не указывать аргументы типа — ошибка. Кстати, если тип принимает несколько аргументов типа, вы должны добавить запятые — например, `typeof(Dictionary <, >)`. Это необходимо, чтобы избежать неоднозначности, когда существует несколько обобщенных типов с одинаковыми именами, но отличающихся только количеством требуемых параметров типа (это называется *арностью*, или количеством аргументов) — например, `typeof(Func <, >)` и `typeof(Func < , , , >)`. Вы не можете указать частично связанный обобщенный тип. Например, `typeof(Dictionary<string, >)` не скомпилируется.

Вы можете узнать, когда объект `Type` ссылается на обобщенный тип — свойство `IsGenericType` будет возвращать `true` как для `bound`, так и для `unbound`

из листинга 13.10. Вы также можете определить, были ли предоставлены аргументы типа, используя свойство `IsGenericTypeDefinition`, которое вернуло бы `false` и `true` для объектов `Type`, соответствующих `bound` и `unbound` соответственно. Если у вас есть связанный обобщенный тип и вы хотите получить несвязанный тип, из которого он был создан, можно использовать метод `GetGenericTypeDefinition` — вызов этого метода для `bound` приведет к возвращению того же объекта типа, на который ссылается `unbound`.

Взяв объект `Type`, свойство `IsGenericTypeDefinition` которого возвращает `true`, вы можете создать новую связанную версию этого типа, вызвав `MakeGenericType` и передав массив объектов `Type` по одному для каждого аргумента типа.

Из обобщенного типа вы можете извлечь его аргументы типа, воспользовавшись свойством `GenericTypeArguments`. Может удивить, что это работает даже для несвязанных типов, хотя ведет себя при этом немного иначе. Если вы запросите `GenericTypeArguments` у `bound` из листинга 13.10, он вернет массив, содержащий один объект `Type`, который будет таким же, как вы получили бы из `typeof(int)`. Если вы запросите `unbound.GenericTypeArguments`, вы также получите массив, содержащий один `Type`, но на этот раз это будет объект, который не представляет какой-то определенный тип, — его свойство `IsGenericParameter` будет иметь значение `true`, что указывает на заполнитель. Его имя в этом случае будет `T`. Как правило, имя будет соответствовать имени заполнителя по выбору самого обобщенного типа. Например, с помощью `typeof(Dictionary<, >)` вы получите два объекта `Type` с именами `TKey` и `TValue` соответственно. Вы встретите аналогичные обобщенные типы заполнителей аргументов, если будете использовать API отражения для поиска членов обобщенных типов. Например, если вы извлечете `MethodInfo` метода `Add` несвязанного типа `List<>`, вы обнаружите, что он принимает единственный аргумент типа с именем `T`, который возвращает `true` из своего свойства `IsGenericParameter`.

Когда объект `Type` представляет несвязанный обобщенный параметр, вы можете узнать, является ли параметр ковариантным или контравариантным (или ни тем ни другим), с помощью его метода `GenericParameterAttributes`.

MethodBase, ConstructorInfo и MethodInfo

Конструкторы и методы имеют много общего. Для обоих типов членов доступны одинаковые параметры видимости, у обоих есть списки аргументов,

и они могут содержать код. Следовательно, типы отражений `MethodInfo` и `ConstructorInfo` совместно используют базовый класс `MethodBase`, который определяет свойства и методы для обработки этих общих аспектов.

Помимо использования свойств класса `Type`, о которых я говорил ранее, вы можете получить `MethodInfo` или `ConstructorInfo`, вызвав статический метод `GetCurrentMethod` класса `MethodBase`. Он проверяет вызывающий код и определяет, является ли он конструктором или же обычным методом, после чего возвращает либо `MethodInfo`, либо `ConstructorInfo` соответственно.

Помимо членов, которые он наследует от `MemberInfo`, `MethodBase` определяет свойства, определяющие видимость члена. По своей идее они похожи на те, что я описал ранее для типов, но их имена незначительно отличаются, потому что, в отличие от `Type`, `MethodBase` не определяет свойства доступности, которые отличают вложенные и не вложенные элементы. Таким образом, в случае `MethodBase` мы видим `IsPublic`, `IsPrivate`, `IsAssembly`, `IsFamily`, `IsFamilyOrAssembly` и `IsFamilyAndAssembly` для `public`, `private`, `internal`, `protected`, `internal` и `protected private` соответственно.

В дополнение к свойствам, связанным с видимостью, `MethodBase` определяет свойства, которые сообщают о других аспектах метода, такие как `IsStatic`, `IsAbstract`, `IsVirtual`, `IsFinal` и `IsConstructor`.

Есть также и свойства для работы с обобщенными методами. `IsGenericMethod` и `IsGenericMethodDefinition` являются эквивалентами уровня методов для свойств уровня типа `IsGenericType` и `IsGenericTypeDefinition`. Как и в случае с `Type`, имеется метод `GetGenericMethodDefinition` для перехода от связанного обобщенного метода к несвязанному, а также метод `MakeGenericMethod` для создания связанного обобщенного метода из несвязанного. Вы можете получить аргументы типа, вызвав `GetGenericArguments`, и, как и в случае обобщенных типов, он вернет определенные типы при вызове в связанном методе и типы заполнителей при использовании с несвязанным методом.

Вы можете изучить реализацию метода, вызвав `GetMethodBody`. Метод вернет объект `MethodBody`, который дает доступ к IL (в виде массива байтов), а также к определениям локальных переменных, используемых методом.

Класс `MethodInfo` является производным от `MethodBase` и представляет только методы (но не конструкторы). Он добавляет свойство `ReturnType`, которое предоставляет объект `Type`, указывающий на тип возврата метода. (Существует специальный системный тип `System.Void`, объект `Type` которого используется, если метод ничего не возвращает.)

Класс `ConstructorInfo` не добавляет никаких свойств, кроме тех, которые наследует от `MethodBase`. Однако он определяет два статических поля только для чтения: `ConstructorName` и `TypeConstructorName`. Они содержат строки `.ctor` и `.cctor` соответственно, которые являются значениями, которые вы найдете, например, в свойстве `Name` объектов `ConstructorInfo` и статических конструкторов. Если речь о CLR, то это настоящие имена — хотя в C# конструкторы имеют то же имя, что и содержащий их тип, это присутствует только в исходных файлах C#, но не во время выполнения.

Вы можете вызвать метод или конструктор, представленный `MethodInfo` или `ConstructorInfo`, с помощью `Invoke`. Он делает то же самое, что и `Type.InvokeMember` — он использовался в листинге 13.9 для вызова метода. Однако поскольку `Invoke` специализируется на работе с методами и конструкторами, использовать его гораздо проще. В случае с `ConstructorInfo` требуется передать только массив аргументов. Используя `MethodInfo`, вы дополнительно передаете объект, для которого хотите вызвать метод, или `null`, если требуется вызвать статический метод. Листинг 13.11 делает то же самое, что и листинг 13.9, но с использованием `MethodInfo`.

Листинг 13.11. Вызов метода

```
public static object CreateAndInvokeMethod(
    string typeName, string member, params object[] args)
{
    Type t = Type.GetType(typeName);
    object instance = Activator.CreateInstance(t);
    MethodInfo m = t.GetMethod(member);
    return m.Invoke(instance, args);
}
```

Как для методов, так и для конструкторов вы можете вызвать метод `GetParameters`, который возвращает массив объектов `ParameterInfo`, представляющих параметры метода.

ParameterInfo

Класс `ParameterInfo` представляет параметры для методов или конструкторов. Его свойства `ParameterType` и `Name` дают основную информацию, которую вы видите в сигнатуре метода. Он также определяет свойство `Member`, ссылающееся на метод или конструктор, которому принадлежит параметр. Свойство `HasDefaultValue` сообщает вам, является ли параметр необязатель-

ным, и, если это так, `DefaultValue` предоставляет значение, которое будет использоваться, когда аргумент не передается.

Если вы работаете с членами, определенными несвязанными обобщенными типами или с несвязанным обобщенным методом, имейте в виду, что `ParameterType` для `ParameterInfo` может ссылаться на аргумент обобщенного типа, а не на реальный тип. Это также верно для любых объектов `Type`, возвращаемых объектами отражения, которые описаны в следующих трех подразделах.

FieldInfo

`FieldInfo` представляет поле в типе. Обычно вы получаете его из объекта `Type` с помощью `GetField` или `GetFields` или, если используете код на языке с поддержкой глобальных полей, вы можете извлечь его из содержащего модуля.

`FieldInfo` определяет набор свойств, представляющих видимость. Они выглядят так же, как те, которые определены в `MethodBase`. Кроме того, существует метод `FieldType`, представляющей тип, который может содержать поле. (Как обычно, если член принадлежит несвязанному обобщенному типу, оно может относиться к аргументу типа, а не к конкретному типу.) Существует также ряд свойств, представляющих дополнительную информацию о поле, таких как `IsStatic`, `IsInitOnly` и `IsLiteral`. Они соответствуют `static`, `readonly` и `const` в C# соответственно. (`IsLiteral` также возвращает `true` для полей, представляющих значения в типах перечисления.)

`FieldInfo` определяет методы `GetValue` и `SetValue`, которые позволяют вам прочитать и записать значение поля. Они принимают аргумент, указывающий используемый экземпляр, или `null`, если поле является статическим. Как и в случае `Invoke` класса `MethodBase`, они не делают ничего, чего вы не могли бы добиться с помощью `InvokeMember` класса `Type`, но эти методы, как правило, более удобны.

PropertyInfo

Тип `PropertyInfo` представляет свойство. Вы можете получить их из метода `GetProperty` или `GetProperties` содержащего объекта `Type`. Как я упоминал ранее, `PropertyInfo` не определяет свойств для видимости, потому что видимость определяется на уровне отдельных методов доступа. Вы можете получить их с помощью методов `GetGetMethod` и `GetSetMethod`, оба из которых возвращают объекты `MethodInfo`.

Как и в случае с `FieldInfo`, класс `PropertyInfo` определяет методы `GetValue` и `SetValue` для чтения и записи значения. Свойства могут принимать аргументы. Например, индексаторы C# – это свойства с аргументами. Следовательно, существуют перегрузки `GetValue` и `SetValue`, которые принимают массивы аргументов. Кроме того, существует метод `GetIndexParameters`, который возвращает массив объектов `ParameterInfo`, представляющих аргументы, которые необходимы для использования свойства. Тип свойства доступен через свойство `PropertyType`.

EventInfo

События представлены объектами `EventInfo`, которые возвращаются методами `GetEvent` и `GetEvents` класса `Type`. Как и в случае `PropertyInfo`, он не имеет никаких свойств видимости, так как методы добавления и удаления события самостоятельно определяют собственную видимость. Вы можете получить эти методы с помощью `GetAddMethod` и `GetRemove Method`, оба из которых возвращают `MethodInfo`. `EventInfo` определяет `EventHandlerType`, возвращающий тип делегата, который требуется предоставить обработчикам событий.

Вы можете прикреплять и удалять обработчики, вызывая методы `AddEventHandler` и `RemoveEventHandler`. Как и в случае с другими динамическими вызовами, они являются всего лишь более удобной альтернативой метода `InvokeMember` класса `Type`.

Контексты отражения

Платформа .NET содержит функцию, называемую *контекстами отражения*. Они позволяют отражению предоставлять виртуализированное представление системы типов. Путем написания пользовательского контекста отражения вы можете изменить способ отображения типов – можно сделать так, чтобы тип выглядел, как будто у него есть дополнительные свойства, или дополнить набор атрибутов, которые предлагают члены и параметры. (Атрибуты описываются в главе 14.)

Контексты отражения полезны, потому что позволяют создавать управляемые отражением структуры, которые дают отдельным типам возможность настраивать способ их обработки, не принуждая при этом каждый участвующий в этом тип предоставить явную поддержку. До введения

пользовательских контекстов отражения в .NET 4.5 это обрабатывалось различными специальными системами. Возьмем, к примеру, панель **Properties** в Visual Studio. Она способна автоматически отображать каждое публичное свойство, определенное любым объектом .NET в области проектирования (например, любым компонентом пользовательского интерфейса, который вы пишете). Прекрасно, когда есть поддержка автоматического редактирования даже для компонентов, которые не обеспечивают какой-либо явной обработки этого, но компоненты должны иметь возможность настраивать свое поведение во время разработки.



Контексты отражения не работали в .NET Core до версии 3.0. Код, который их использует, можно скомпилировать благодаря наличию пакета `NuGet System.Reflection.Context`, из-за чего можно подумать, что они работают. Однако изначально эта возможность появилась для облегчения переноса кода с .NET Framework на .NET Core и .NET Standard. Это позволило писать библиотеки .NET Standard, использующие пользовательские контексты отражения, и, если бы вы использовали эти библиотеки в .NET Framework, все было бы хорошо. Если бы вы использовали эти библиотеки в .NET Core так, чтобы не касаться кода, пытающегося использовать пользовательские контексты отражения, все бы тоже было хорошо. Проблемы в .NET Core неизбежны, только если пытаться использовать функцию, основанную на пользовательских контекстах отражений. В этом случае вас ждет `PlatformNotSupportedException`. Но с .NET Core 3.0 и v4.6 или новее, из пакета `NuGet System.Reflection.Context` все будет отлично работать, потому что с .NET Core 3.0 в CLR добавились необходимые возможности.

Поскольку панель **Properties** предшествует .NET 4.5, она использует одно из этих специальных решений: класс `TypeDescriptor`. Это обертка поверх отражения, которая позволяет любому классу расширять свое поведение во время разработки путем реализации `ICustomTypeDescriptor`. Это дает классу возможность настраивать набор предлагаемых для редактирования свойств, а также контролировать их представление, предлагая даже пользовательское редактирование интерфейсов. Это довольно гибко, но имеет недостаток при связывании кода времени разработки с кодом времени выполнения — компоненты, использующие эту модель, не могут поставляться без предоставления кода времени разработки. Поэтому Visual Studio ввела свои собственные механизмы виртуализации для их разделения.

Чтобы каждая среда не определяла собственную систему виртуализации, пользовательские контексты отражения добавляют виртуализацию непосредственно в API отражения. Если вам нужен код, который может использовать информацию о типе, предоставленную отражением, а также поддерживать дополнение или модификацию этой информации во время разработки, вам больше не нужно использовать специальную обертку. Можно использовать обычные типы отражений, описанные ранее в этой главе, но при этом попросить, чтобы отражение предоставило вам разные реализации этих типов посредством разных виртуализированных представлений.

Это делается путем написания пользовательского контекста отражения, который описывает, как нужно изменить представление, предоставляемое отражением. В листинге 13.12 показан ничем не примечательный тип, за которым следует пользовательский контекст отражения, делающий этот тип похожим на свойство.

Листинг 13.12. Простой тип, расширенный контектом отражения

```
class NotVeryInteresting
{ }

class MyReflectionContext : CustomReflectionContext
{
    protected override IEnumerable< PropertyInfo> AddProperties(Type type)
    {
        if (type == typeof(NotVeryInteresting))
        {
            var fakeProp = CreateProperty(
                MapType(typeof(string).GetTypeInfo()),
                "FakeProperty",
                o => "FakeValue",
                (o, v) => Console.WriteLine($"Setting value: {v}"));

            return new[] { fakeProp };
        }
        else
        {
            return base.AddProperties(type);
        }
    }
}
```

Код, который напрямую использует API отражения, увидит тип `NotVeryInteresting` непосредственно как он есть, без каких-либо свойств. Однако мы можем отобразить этот тип через `MyReflectionContext`, как показано в листинге 13.13.

Листинг 13.13. Использование пользовательского контекста отражения

```
var ctx = new MyReflectionContext();
TypeInfo mappedType =
    ctx.MapType(typeof(NotVeryInteresting).GetTypeInfo());

foreach ( PropertyInfo prop in mappedType.DeclaredProperties )
{
    Console.WriteLine($"{prop.Name} ({prop.PropertyType.Name})");
}
```

Переменная `mappedType` содержит ссылку на итоговый сопоставленный тип. Он по-прежнему выглядит как обычный объект отражения `TypeInfo`, и мы можем перебирать его свойства обычным способом с помощью `DeclaredProperties`, но, поскольку мы отобразили тип через мой пользовательский контекст отражения, мы видим измененную версию типа. Вывод этого кода покажет, что тип определяет одно свойство с именем `FakeProperty` типа `string`.

Итог

API отражения позволяет писать код, поведение которого основано на структуре типов, с которыми он работает. Это может включать в себя решение на основе свойств объекта о том, какие значения показывать в сетке пользовательского интерфейса, или это может означать изменение поведения платформы на основе того, какие члены определяет конкретный тип. Например, части веб-платформы ASP.NET Core смогут определить, использует ли ваш код методы синхронного или асинхронного программирования, после чего соответствующим образом адаптируется. Эти методы требуют возможности изучить код во время выполнения, что позволяет сделать отражение. Вся информация в сборке, которая требуется системе типов, доступна и для нашего кода. Более того, вы можете представить ее в виртуализированном виде, написав собственный контекст отражения, позволяющий настроить поведение кода, управляемого отражением.

ГЛАВА 14

Атрибуты

В .NET можно аннотировать компоненты, типы и их члены с помощью атрибутов. Назначение атрибута — регулировать или изменять поведение платформы, инструмента, компилятора или CLR. Например, в главе 1 я демонстрировал класс, аннотированный атрибутом `[TestClass]`. Он сообщал инфраструктуре юнит-теста, что класс содержит ряд тестов, которые должны быть выполнены как часть набора тестов.

Атрибуты лишь содержат информацию, но ничего не делают сами по себе. Проведу аналогию с физическим миром. Если вы распечатываете этикетку, содержащую информацию о месте назначения и отслеживании, и прикрепляете ее к посылке, эта этикетка сама по себе не заставит посылку добраться до пункта назначения. Такая этикетка полезна только тогда, когда груз находится в обработке у транспортной компании. Когда компания заберет посылку, этикетка понадобится, чтобы определить, как и куда направить груз. Таким образом, этикетка важна, но ее единственной задачей является предоставление информации, которая требуется какой-либо системе. С атрибутами .NET все так же — они работают только в том случае, если кто-то ожидает их найти. Некоторые атрибуты обрабатываются CLR или компилятором, но таких меньшинство. Большинство же используются платформами, библиотеками, инструментами (например, системами юнит-теста) или вашим собственным кодом.

Применение атрибутов

Во избежание необходимости вводить дополнительный набор понятий в систему типов .NET работает с ними как с экземплярами типов .NET. Для использования в качестве атрибута тип должен быть производным от класса `System.Attribute`, и это его единственная особенность. Чтобы применить атрибут, вы помещаете имя типа в квадратные скобки и, как правило, размещаете непосредственно перед целью атрибута. Листинг 14.1 показывает некоторые атрибуты из среды тестирования Microsoft. Один

я применил к классу, чтобы указать, что он содержит тесты, которые я хотел бы запустить. Кроме этого, я применил атрибуты к отдельным методам, сообщая среде тестирования, какие из них представляют собой тесты, а какие содержат код инициализации, который должен выполняться перед каждым тестом.

Листинг 14.1. Атрибуты в классе юнит-теста

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ImageManagement.Tests
{
    [TestClass]
    public class WhenPropertiesRetrieved
    {
        private ImageMetadataReader _reader;

        [TestInitialize]
        public void Initialize()
        {
            _reader = new ImageMetadataReader(TestFiles.GetImage());
        }

        [TestMethod]
        public void ReportsCameraMaker()
        {
            Assert.AreEqual(_reader.CameraManufacturer, "Fabrikam");
        }

        [TestMethod]
        public void ReportsCameraModel()
        {
            Assert.AreEqual(_reader.CameraModel, "Fabrikam F450D");
        }
    }
}
```

Если вы заглянете в документацию по большинству атрибутов, вы обнаружите, что их настоящие имена оканчиваются на **Attribute**. Если нет класса с именем, указанным в скобках, компилятор C# попытается добавить **Attribute**, поэтому атрибут **[TestClass]** в листинге 14.1 ссылается на класс **TestClassAttribute**. Если хотите, вы можете записывать имя класса полностью, например **[TestClassAttribute]**, но чаще используют более короткую форму.

Если необходимо применить несколько атрибутов, то у вас есть два варианта. Можно предоставить несколько наборов скобок или поместить несколько атрибутов в одну пару скобок, разделив их запятыми.

Некоторые типы атрибутов способны принимать аргументы конструктора. Например, среда тестирования Microsoft содержит атрибут `TestCategory-Attribute`. При запуске можно выбрать выполнение только тех тестов, которые принадлежат определенной категории. Данный атрибут требует, чтобы вы передали имя категории в качестве аргумента конструктора, потому что нет смысла применять этот атрибут без указания имени. Синтаксис для указания аргументов конструктора атрибута вполне ожидаемый (листинг 14.2).

Листинг 14.2. Атрибут с аргументом конструктора

```
[TestCategory("Property Handling")]
[TestMethod]
public void ReportsCameraMaker()
{
    ...
}
```

Вы также можете указать свойства или значения полей. Характеристиками некоторых атрибутов можно управлять только через свойства или поля, но не через аргументы конструктора. (Если атрибут имеет множество необязательных настроек, обычно проще представить их как свойства или поля вместо определения перегрузки конструктора для каждой возможной комбинации настроек.) Синтаксис заключается в одной или нескольких записях вида `PropertyName=PropertyValue` после аргументов конструктора (или вместо них, если их нет). В листинге 14.3 показан другой атрибут, используемый в юнит-тестировании, `ExpectedExceptionAttribute`, позволяющий указать, что при выполнении теста вы ожидаете, что он выдаст конкретное исключение. Тип исключения является обязательным, поэтому мы передаем его в качестве аргумента конструктора, но данный атрибут позволяет также указать, должен ли исполнитель теста принимать исключения типа, производного от указанного. (По умолчанию он принимает только точное совпадение.) Это поведение управляетя с помощью свойства `AllowDerivedTypes`.

Листинг 14.3. Указание необязательных настроек атрибута со свойствами

```
[ExpectedException(typeof(ArgumentException), AllowDerivedTypes = true)]
[TestMethod]
public void ThrowsWhenNameMalformed()
{
    ...
}
```

Применение атрибута не ведет к его созданию. Все, что вы делаете при применении атрибута, — это предоставляет инструкции о том, как атрибут должен быть создан и инициализирован, если он кому-то потребуется. (Бытует распространенное заблуждение, что атрибуты метода создаются при запуске метода. Это не так.) Когда компилятор создает метаданные для сборки, он включает информацию о том, какие атрибуты были применены к каким элементам, включая список аргументов конструктора и значения свойств, а CLR будет извлекать и использовать эту информацию, только если это кому-то понадобится. Например, когда вы говорите Visual Studio запустить ваши юнит-тесты, он загрузит вашу тестовую сборку, а затем для каждого открытого типа запросит у CLR все связанные с тестиированием атрибуты. Это и есть та точка, в которой создаются атрибуты. Если просто загрузить сборку, скажем добавив ссылку на нее из другого проекта, а затем использовать некоторые из содержащихся в ней типов, то атрибуты создаваться не будут — они останутся не более чем набором инструкций, затерянных в метаданных вашей сборки.

Цели атрибутов

Атрибуты могут применяться ко множеству различных типов целей. Вы можете поместить атрибуты в любую из функций системы типов, представленных в API отражения из главы 13. В частности, их можно применять атрибуты к сборкам, модулям, типам, методам, параметрам методов, конструкторам, полям, свойствам, событиям и параметрам обобщенного типа. Кроме того, вы можете предоставить атрибуты, цель которых — возвращаемое значение метода.

В большинстве случаев вы обозначаете цель, просто помещая атрибут непосредственно перед ней. Но это не сработает в случае сборок или модулей, потому что в вашем исходном коде нет ничего, что бы их представляло, — все в вашем проекте идет в сборку, которую он производит. Модули, в свою очередь, тоже являются совокупностью (как правило, составляя сборку, как я описал в главе 12). Поэтому для них мы должны явно указать цель в начале атрибута. Вы часто будете видеть атрибуты уровня сборки, подобные показанным в листинге 14.4, в файле `GlobalSuppressions.cs`. Visual Studio иногда предлагает варианты для изменения вашего кода, и если вы решите подавить этот функционал, это можно сделать с помощью атрибутов уровня сборки.

Вы можете поместить атрибуты уровня сборки в любой файл. Единственное ограничение заключается в том, что они должны появляться перед определением пространства имен или типа. Все, что должно предшествовать атрибутам уровня сборки, — это нужные вам директивы `using`, комментарии и пробелы (все это необязательно).

Листинг 14.4. Атрибуты уровня сборки

```
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage(
    "StyleCop.CSharp.NamingRules",
    "SA1313:Parameter names should begin with lower-case letter",
    Justification = "Triple underscore acceptable for unused lambda
        parameter",
    Scope = "member",
    Target = "~M:Idg.Examples.SomeMethod")]
```

Атрибуты уровня модуля следуют той же схеме, хотя и встречаются гораздо реже. Не в последнюю очередь это происходит потому, что многомодульные сборки встречаются довольно редко и не поддерживаются .NET Core. В листинге 14.5 показано, как настроить возможность отладки конкретного модуля в том случае, если вы хотите, чтобы один модуль в многомодульной сборке был легко отлаживаемым, а остальные — JIT-компилируемыми с полной оптимизацией. (Это специально придуманный сценарий, с помощью которого я могу показать синтаксис. На практике вы вряд ли захотите это делать.) Я расскажу об атрибуте `DebuggableAttribute` позже, в подразделе «JIT-компиляция» на с. 743.

Листинг 14.5. Атрибут уровня модуля

```
using System.Diagnostics;

[module: Debuggable(
    DebuggableAttribute.DebuggingModes.DisableOptimizations)]
```

Возвращаемые значения методов могут быть аннотированы, и это также требует квалификации, потому что атрибуты возвращаемого значения располагаются перед методом, там же, где и атрибуты, которые применяются к самому методу. (Атрибуты для параметров не нуждаются в квалификации, потому что они располагаются в круглых скобках вместе с аргументами.) В листинге 14.6 показан метод с атрибутами, применяемыми как к методу, так и к типу возвращаемого значения. (Атрибуты в этом примере являются частью служб взаимодействия, которые позволяют коду .NET вызывать внешний код, такой как API ОС. В этом примере импортируется функция

библиотеки Win32, что позволяет использовать ее из C#. Существует несколько различных представлений для логических значений в неуправляемом коде, поэтому в данном случае я аннотировал возвращаемый тип с помощью атрибута `MarshalAsAttribute`, указав, какой именно тип следует ожидать CLR.)

Листинг 14.6. Атрибуты метода и возвращаемого значения

```
[DllImport("User32.dll")]
[return: MarshalAs(UnmanagedType.Bool)]
static extern bool IsWindowVisible(HandleRef hWnd);
```

Другой вид цели, которая нуждается в квалификации, — это поля, генерируемые компилятором. Вы получаете их вместе со свойствами, для которых вы не предоставили код для метода получения или установки, а также в случае членов событий без явной реализации `add` и `remove`. Атрибуты в листинге 14.7 применяются к полям, которые содержат значение свойства и делегат для события; без квалификаторов `field`: атрибуты в этих позициях будут применяться к самому свойству или событию.

Листинг 14.7. Атрибут для генерируемых компилятором полей свойств и событий

```
[field: NonSerialized]
public int DynamicId { get; set; }

[field: NonSerialized]
public event EventHandler Frazzled;
```

Атрибуты, обрабатываемые компилятором

Компилятор C# распознает определенные типы атрибутов и обрабатывает их особым образом. Например, имена сборок и версии задаются с помощью атрибутов, как и другая связанная информация о вашей сборке. Как описано в главе 12, в современных проектах .NET процесс сборки генерирует скрытый исходный файл, в котором все это и содержится. Если вам любопытно, он обычно отправляется в папку `obj\Debug` или `obj\Release` вашего проекта и получает имя вроде `YourProject.AssemblyInfo.cs`. В листинге 14.8 приведен типичный пример.

В проектах, создававшихся до появления .NET Core, процесс сборки автоматически не генерировал этот файл, поэтому вместо него большинство проектов включали в себя файл `AssemblyInfo.cs` (хотя по умолчанию Visual Studio прятал его в узле `Properties` проекта в обозревателе решений).

Листинг 14.8. Типичный генерируемый файл с атрибутами уровня сборки

```
//-----
// <auto-generated>
//   This code was generated by a tool.
//   Runtime Version:4.0.30319.42000
//
//   Changes to this file may cause incorrect behavior and will be lost if
//   the code is regenerated.
// </auto-generated>
//-----

using System;
using System.Reflection;

[assembly: System.Reflection.AssemblyCompanyAttribute("MyCompany")]
[assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
[assembly: System.Reflection.AssemblyFileVersionAttribute("1.0.0.0")]
[assembly: System.Reflection.AssemblyInformationalVersionAttribu
te("1.0.0")]
[assembly: System.Reflection.AssemblyProductAttribute("MyApp")]
[assembly: System.Reflection.AssemblyTitleAttribute("MyApp")]
[assembly: System.Reflection.AssemblyVersionAttribute("1.0.0.0")]

// Создается классом MSBuild WriteCodeFragment.
```

Даже если вы управляете этими атрибутами лишь косвенно, их полезно понимать, поскольку они влияют на выходные данные компилятора.

Имена и версии

Как вы видели в главе 12, сборки имеют составное имя. Простое имя, которое обычно совпадает с именем файла, но без расширения .exe или .dll, настраивается в рамках параметров проекта. Имя также включает номер версии, и это контролирует атрибут, показанный в листинге 14.9.

Листинг 14.9. Атрибуты версии

```
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Как вы, возможно, помните из главы 12, первый задает часть имени сборки, относящуюся к версии. Второй не имеет отношения к .NET — компилятор использует его для генерации ресурса версии в стиле Win32. Это номер версии, который увидят конечные пользователи, если выберут вашу сборку в проводнике Windows и откроют окно Properties.

Культура также является частью имени сборки. Она часто устанавливается автоматически, если вы используете вспомогательные механизмы сборки ресурсов, описанные в главе 12. Культуру можно установить и явно, с помощью атрибута `AssemblyCulture`, но для нересурсных сборок культура обычно не устанавливается. (Единственный атрибут уровня сборки, связанный с культурой, который обычно указывается явно, — это атрибут `NeutralResourcesLanguageAttribute`, который я показал в главе 12.)

Сборки со строгими именами содержат в имени дополнительный компонент: маркер открытого ключа. Самый простой способ задать строгое имя — это вкладка `Signing` свойств вашего проекта. Однако вы также можете управлять строгими именами из исходного кода, потому что компилятор распознает некоторые предназначенные для этого атрибуты. `AssemblyKeyFileAttribute` принимает имя файла, который содержит ключ. Кроме того, вы можете поместить ключ в хранилище ключей компьютера (которое является частью криптографической системы Windows). Если вы хотите это сделать, то можете использовать `AssemblyKeyNameAttribute`. Наличие любого из этих атрибутов заставит компилятор встроить открытый ключ в сборку и добавить хеш этого ключа в качестве маркера открытого ключа строгого имени. Если файл ключа содержит закрытый ключ, компилятор все равно подпишет вашу сборку. Если же это не так, то компиляции не произойдет до тех пор, пока вы не включите либо отложенное, либо открытое подписывание. Вы можете включить отложенное подписывание, применив атрибут `AssemblyDelaySignAt` с аргументом конструктора `true`. В качестве альтернативы вы можете добавить `<DelaySign>true</DelaySign>` или `<PublicSign>true</PublicSign>` в ваш файл `.csproj`.



Хотя атрибуты, связанные с ключом, обрабатываются компилятором особым образом, он все равно встраивает их в метаданные как обычные атрибуты. Таким образом, если вы используете `AssemblyKeyFileAttribute`, путь к вашему файлу ключа будет виден в итоговом скомпилированном выводе. Это не всегда является проблемой, но вы можете не афишировать подобные подробности и вместо основанного на атрибутах подхода использовать конфигурацию строгих имен на уровне проекта.

Описание и связанные ресурсы

Ресурс версии, создаваемый атрибутом `AssemblyFileVersion`, — это не единственная информация, которую компилятор C# способен встраивать в ресурсы в стиле Win32. Есть несколько других атрибутов, предоставляю-

щих информацию об авторских правах и другой описательный текст. Листинг 14.10 показывает типичную выборку таких атрибутов.

Листинг 14.10. Типичные атрибуты описания сборки

```
[assembly: AssemblyTitle("ExamplePlugin")]
[assembly: AssemblyDescription("An example plug-in DLL")]
[assembly: AssemblyConfiguration("Retail")]
[assembly: AssemblyCompany("Endjin Ltd.")]
[assembly: AssemblyProduct("ExamplePlugin")]
[assembly: AssemblyCopyright("Copyright © 2019 Endjin Ltd.")]
[assembly: AssemblyTrademark("")]
```

Как и в случае с версией файла, все они отображаются на вкладке **Details** окна **Properties**, которое Windows Explorer показывает для файла. Также вы можете создать все эти атрибуты, либо напрямую отредактировав файл проекта, либо использовав для этого страницу свойств проекта в Visual Studio.

Атрибуты информации о вызывающем компоненте

Есть ряд обрабатываемых компилятором атрибутов, которые разработаны для сценариев, где вашим методам нужна информация о контексте, из которого они были вызваны. Это может пригодиться в определенных сценариях ведения журнала диагностики, а также полезно при реализации одного интерфейса, который обычно используют в коде пользовательского интерфейса.

В листинге 14.11 показано, как можно использовать эти атрибуты в коде журналирования. Если вы аннотируете параметры метода одним из этих трех новых атрибутов, компилятор обработает их особым образом в тех случаях, когда вызывающие компоненты опускают аргументы. Он передаст либо имя члена (метод или свойство), который вызывает метод с атрибутом, либо имя файла, содержащего вызвавший метод код, либо номер строки, из которой был сделан вызов.

Листинг 14.11. Применение атрибутов информации о вызывающем компоненте к параметрам метода

```
public static void Log(
    string message,
    [CallerMemberName] string callingMethod = "",
    [CallerFilePath] string callingFile = "",
    [CallerLineNumber] int callingLineNumber = 0)
{
    Console.WriteLine(
        "Message {0}, called from {1} in file '{2}', line {3}",
        message, callingMethod, callingFile, callingLineNumber);
}
```

Если вы при вызове этого метода предоставите все аргументы, ничего особенного не произойдет. Но если вы опустите любой из необязательных аргументов, C# создаст код, предоставляющий информацию о месте, откуда был вызван метод. Значениями по умолчанию для трех необязательных аргументов в листинге 14.11 будут имя метода или свойства вызвавшего этот метод `Log`, полный путь к исходному коду, содержащему вызов, и номер строки, из которой `Log` был вызван.



Эти атрибуты разрешены только для необязательных параметров. Единственный способ сделать аргумент необязательным — предоставить значение по умолчанию для этого аргумента. При вызове из C# (или из Visual Basic, который также поддерживает эти атрибуты) и при наличии этих атрибутов C# всегда будет заменять значение по умолчанию, поэтому оно никогда не будет использоваться. Тем не менее вы должны указать значение по умолчанию, поскольку без него параметр будет считаться обязательным, так что мы обычно используем пустые строки, `null` или число 0.

Атрибут `CallerMemberName` напоминает оператор `nameof`, с которым мы встречались в главе 8. И тот и другой заставляют компилятор создать строку, содержащую имя некоторой функции кода, но работают они совершенно по-разному. В случае `nameof` всегда известно, какую строку вы получите, потому что она определяется предоставляемым выражением. (Например, `nameof (message)` внутри `Log` в листинге 14.11 всегда будет иметь результат `message`.) Но `CallerMemberName` меняет способ, которым компилятор вызывает метод, к которому они применяются, — `callMethod` имеет этот атрибут, и его значение не является фиксированным. Оно будет зависеть от того, откуда вызывается этот метод.



Обнаружить вызывающий метод можно другим способом: классы `StackTrace` и `StackFrame` в пространстве имен `System.Diagnostics` сообщают информацию о методах, расположенных выше в стеке вызовов. Однако они имеют значительно более высокие затраты в плане времени выполнения, так как атрибуты информации о вызывающем компоненте вычисляют значения во время компиляции, что делает издержки времени выполнения очень низкими. (Как и в случае с `nameof`.) Кроме того, `StackFrame` способен определять имя файла и номер строки, только если доступны символы отладки.

Хотя диагностическое ведение журнала является очевидной областью применения данной функции, я также упомянул определенный сценарий, с которым сталкиваются большинство разработчиков пользовательского интерфейса .NET. Библиотека классов .NET определяет интерфейс `IPropertyChanged`. Как показано в листинге 14.12, это очень простой интерфейс, который содержит единственный член — событие `PropertyChanged`.

Листинг 14.12. `IPropertyChanged`

```
public interface IPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

Типы, которые реализуют этот интерфейс, вызывают событие `PropertyChanged` каждый раз, когда изменяется какое-то из их свойств. `PropertyChangedEventArgs` предоставляет строку, содержащую имя только что измененного свойства. Эти уведомления об изменениях полезны при работе с пользовательским интерфейсом, потому что позволяют использовать объект с технологиями привязки данных (например, предоставляемыми платформой пользовательского интерфейса WPF .NET), которые способны автоматически обновлять пользовательский интерфейс при каждом изменении свойства.

Привязка данных может помочь вам добиться четкого разделения между кодом, который непосредственно связан с типами пользовательского интерфейса, и кодом, который содержит логику того, как приложение должно реагировать на ввод пользователя.

Реализация `IPropertyChanged` довольно утомительна и может привести к ошибкам. Поскольку событие `PropertyChanged` в виде строки указывает, какое свойство изменилось, очень легко напечатать неверное имя свойства или случайно использовать неправильное имя при копировании и вставке реализации из одного свойства в другое. Кроме того, если вы переименовываете свойство, легко забыть изменить текст, используемый с событием, а это означает, что код, который ранее был правильным, теперь будет выдавать неправильное имя при срабатывании события `PropertyChanged`. Оператор `nameof` помогает в случае опечаток и неверного переименования, но не всегда может указать на ошибки копирования. (Например, он не отреагирует на обновление имени при вставке кода из свойства в свойство одного и того же класса.)

Атрибуты информации о вызывающем компоненте могут значительно снизить шанс появления ошибок в реализации этого интерфейса. Листинг 14.13 демонстрирует базовый класс, который реализует `INotifyPropertyChanged`, что дает возможность получать уведомления об изменениях с помощью одного из этих атрибутов. (Он также использует `null`-условный оператор `?,`, чтобы гарантировать вызов делегата события, только если тот не равен `null`. Кстати, когда вы используете оператор таким образом, C# генерирует код, который вычисляет аргументы метода делегата `Invoke`, только если он не равен `null`. Таким образом, он не только опускает вызов `Invoke`, если делегат имеет значение `null`, но и избегает создания `PropertyChangedEventArgs`, который был бы передан в качестве аргумента.) Этот код также определяет, действительно ли значение изменилось, вызывая событие, только если это так. Его возвращаемое значение указывает, изменилось ли значение (на случай, если это понадобится вызывающему компоненту).

Листинг 14.13. Повторно используемая реализация `INotifyPropertyChanged`

```
public class NotifyPropertyChanged : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected bool SetProperty<T>(
        ref T field,
        T value,
        [CallerMemberName] string propertyName = null)
    {
        if (!Equals(field, value))
        {
            return false;
        }

        field = value;
        PropertyChanged?.Invoke(
            this, new PropertyChangedEventArgs(propertyName));
        return true;
    }
}
```

Наличие атрибута `[CallerMemberName]` означает, что классу, производному от этого типа, не нужно указывать имя свойства, если он вызывает `SetProperty` из установщика свойства, как показано в листинге 14.14.

Листинг 14.14. Запуск события изменения свойства

```
public class MyViewModel : NotifyPropertyChanged
{
    private string _name;

    public string Name
    {
        get => _name;
        set => SetProperty(ref _name, value);
    }
}
```

Даже с новым атрибутом реализация `INotifyPropertyChanged` явно требует больших усилий, нежели автоматическое свойство, где вы просто пишете `{get; set;}` и позволяете компилятору сделать за вас всю работу. Но это лишь немного сложнее реализации тривиального свойства с полями и проще, чем было бы возможно без атрибута `[CallerMemberName]`, потому что я смог опустить имя свойства при запросе к базовому классу касательно вызова события. Что еще более важно, снизилась и возможность допустить ошибку: теперь я могу быть уверен, что каждый раз будет использовано правильное имя, даже если в будущем я переименую свойство.

Атрибуты, обрабатываемые CLR

Некоторые атрибуты во время выполнения обрабатываются средой CLR особым образом. Официального исчерпывающего списка таких атрибутов нет, поэтому в следующих нескольких разделах я просто опишу некоторые широко используемые.

InternalsVisibleToAttribute

Вы можете применить `InternalsVisibleToAttribute` к сборке, чтобы объявить, что любые внутренние типы или элементы, которые она определяет, должны быть видимы для одной или нескольких других сборок. Популярная область применения для этого — это юнит-тест внутренних типов. Как показано в листинге 14.15, нужно просто передать имя сборки в качестве аргумента конструктора.

Листинг 14.15. InternalsVisibleToAttribute

```
[assembly:InternalsVisibleTo("ImageManagement.Tests")]
[assembly:InternalsVisibleTo("ImageServices.Tests")]
```



Строгое именование усложняет ситуацию. Сборки со строгими именами не могут сделать свои внутренние компоненты видимыми для сборок с нестрогими именами, и наоборот. Когда сборка со строгим именем делает свои внутренние элементы видимыми для другой сборки со строгим именем, она должна указывать не только простое имя, но и открытый ключ сборки, доступ к которой предоставляется. И это не просто маркер открытого ключа, который я описал в главе 12, — это шестнадцатеричный код всего открытого ключа из нескольких сотен цифр. Вы можете получить полный открытый ключ сборки с помощью утилиты sn.exe в .NET SDK, используя ключ -Tr и путь к сборке.

Пример показывает, что можно сделать типы видимыми для нескольких сборок, применяя атрибут несколько раз с разными именами сборок.

CLR отвечает за соблюдение правил видимости. При обычных обстоятельствах, если вы попытаетесь использовать внутренний класс из другой сборки, то получите ошибку во время выполнения. (C# даже не позволит вам скомпилировать такой код, но компилятор можно обмануть. Или можно писать прямо на IL. IL-ассемблер ILASM сделает все, что вы хотите, и при этом наложит гораздо меньше ограничений, чем C#. Как только вы преодолеете ограничения времени компиляции, то попадете под ограничения времени выполнения.) Но, когда этот атрибут присутствует, CLR ослабляет свои правила для указанных сборок. Компилятор также понимает этот атрибут и позволяет коду, который пытается использовать определенные во внешнем коде внутренние типы, компилировать, пока внешняя библиотека упоминает вашу сборку в `InternalsVisibleToAttribute`.

Этот атрибут обеспечивает лучшее решение проблемы, с которой я столкнулся в первом примере в главе 1, — я хотел изнутри теста использовать точку входа в программу, но по умолчанию содержащий ее класс `Program` является внутренним. Я исправил это, сделав и его, и метод `Main` публичными, но если бы вместо этого я использовал атрибут `InternalsVisibleTo`, то мог бы оставить класс `internal`. Пришлось бы сделать `Main` более видимым — по умолчанию он является `private`, так что мне нужно было сделать его хотя бы `internal`, что все равно лучше, чем делать его `public`.

Помимо того что атрибут полезен в сценариях юнит-теста, его также можно использовать, если вы хотите распределить код по нескольким сборкам. Если вы написали большую библиотеку классов, вы можете не захотеть помещать ее в одну массивную DLL. Если у нее есть несколько разделов, которые ваши

клиенты могут пожелать использовать изолированно, возможно, имеет смысл разделить библиотеку так, чтобы они могли развертывать только необходимые фрагменты. Однако, несмотря на то что вы можете разделить публичный API вашей библиотеки, реализация может оказаться не такой простой, особенно если ваша кодовая база часто использует код повторно. У вас может быть много классов, которые не предназначены для общего использования, но которые используются по всему коду.

Если бы не `InternalsVisibleToAttribute`, было бы крайне неудобно повторно использовать общие детали реализации в различных сборках. Либо каждая сборка должна содержать собственную копию соответствующих классов, либо нужно сделать их публичными типами в составе какой-то общей сборки. Проблема со вторым методом заключается в том, что установка видимости типов как `public` по сути поощряет людей их использовать. Можно указать в документации, что типы предназначены для внутреннего использования вашей платформой, но это остановит далеко не всех.

К счастью, теперь не нужно делать их `public`. Любые типы, которые являются просто деталями реализации, могут оставаться `internal`, и вы можете сделать их доступными для всех ваших сборок с помощью `InternalsVisibleToAttribute`, сохраняя при этом их недоступность для всех остальных.

Атрибуты сериализации

CLR может *сериализовать* определенные объекты, что означает, что он может записывать все значения в полях объекта в двоичный поток. Через некоторое время он может *десериализовать* этот поток обратно в новый объект, возможно, в другом процессе или даже на другом компьютере. Когда сериализация встречает поля, содержащие ссылки, она автоматически сериализует и объекты, на которые вы ссылаетесь. Во избежание попадания в бесконечный цикл сериализация обнаруживает циклические ссылки. Эта функция имеет любопытный статус: Microsoft не рекомендует ее использовать, изначально не включала ее в .NET Core и намеревалась вообще отказаться от нее. Тем не менее она получила частичную отсрочку, появившись в .NET Core 2.1. Вам по-прежнему не рекомендуется ее использовать, и многие из типов библиотек классов .NET, которые поддерживают сериализацию в .NET Framework, не поддерживаются в .NET Core. Однако в некоторых случаях вы должны ее использовать: например, если вам нужна сериализация исключений через границы процесса, то этого можно добиться через данный механизм. Сериализации в этой главе уделено особое внимание, потому что ее атрибуты несколько необычны.

АТРИБУТЫ ИЛИ ПОЛЬЗОВАТЕЛЬСКИЕ АТРИБУТЫ?

Вам может встретиться термин «пользовательский атрибут». Спецификация C# не определяет значение этого термина, но зато это делает спецификация CLI. Она называет пользовательским любой атрибут, который не имеет специальной внутренней обработки в формате метаданных, так что, когда вы видите текст `CustomAttribute` в .NET API, обычно подразумевается это определение. Подавляющее большинство атрибутов, которые вы будете использовать, вписываются в эту категорию, включая большинство атрибутов, определенных библиотекой классов .NET. Даже некоторые атрибуты с обработкой CLR, такие как `InternalsVisibleToAttribute`, согласно этому определению, являются пользовательскими. Исключениями являются атрибуты с внутренней поддержкой в формате файла, такие как `SerializableAttribute`.

В некоторых местах в документации термин используется несколько иным образом: иногда термин *пользовательский атрибут* используется для обозначения типа атрибута, который не поставляется в составе .NET. Другими словами, различие заключается в том, кто написал тип атрибута — вы или Microsoft. И наоборот, в какой-то момент в части документации использовалось более широкое определение, представляющее `StructLayoutAttribute` в качестве примера *пользовательского атрибута*. Этот атрибут является частью служб взаимодействия CLR (которые позволяют вызывать нативный код, такой как API ОС) и, как и `SerializableAttribute`, является одним из очень немногих внутренних типов атрибутов — формат метаданных .NET имеет собственную обработку определенного функционала взаимодействия. В последние годы Microsoft улучшила документацию и теперь по возможности использует термин *пользовательский атрибут* в том же смысле, что и CLI, а также использует термин *псевдоатрибут* для типа с поддержкой встроенных метаданных. Тем не менее вы все равно можете столкнуться с контентом, который не использует эти имена.

Частично причина неопределенности и непоследовательности заключается в том, что в большинстве ситуаций нет реальной технической необходимости вообще делать различие. Если вы пишете инструмент, который работает непосредственно с двоичным форматом для метаданных, очевидно, нужно знать, какие атрибуты поддерживаются форматом напрямую, но для большей части кода эти детали не важны; в C# это, в любом случае, тот же самый синтаксис, о разнице за вас позаботятся компилятор и среда выполнения. В .NET есть несколько механизмов сериализации, и только один из них имеет встроенную поддержку метаданных, но это не оказывает существенного воздействия на их использование. И нет никакой технической разницы между вашим классом, производным от `Attribute`, и аналогичным классом, который поставляется в составе библиотеки классов .NET.

Не все объекты являются сериализуемыми. Например, рассмотрим объект, представляющий сетевое соединение. Что получится, если вы сериализуете его, скопируете получившийся двоичный поток на другой компьютер, а затем десериализуете? Ожидаете ли вы получить объект, который также будет подключен к той же конечной точке, что и исходный? Для многих сетевых протоколов это невозможно. (Возьмем TCP, очень популярный протокол, который лежит в основе HTTP и многих других форм связи. Адреса двух взаимодействующих компьютеров образуют неотъемлемую часть соединения TCP, поэтому если вы переходите на другой компьютер, то вам по определению нужно новое соединение.)

На практике, поскольку ОС предоставляет сетевой стек, объект, представляющий соединение, вероятно, будет иметь числовое поле, содержащее некоторый непрозрачный дескриптор соединения, предоставленный ОС, который не будет работать в другом процессе. Даже в пределах одной машины к этой идеи есть вопросы. В Windows значения дескрипторов обычно относятся к одному процессу. (В некоторых ситуациях есть способы обмена дескрипторами, но для этого нет общего механизма. Помимо всего прочего, одно конкретное числовое значение дескриптора очень часто означает разные вещи в разных процессах, поэтому, даже если вы хотите поделиться дескриптором из своего процесса с другим процессом, он может уже использовать это же значение дескриптора для обращения к чему-то еще. Таким образом, хотя два разных процесса могут быть в состоянии получить дескрипторы для одного и того же, фактические числовые значения этих дескрипторов часто будут разными.) Десериализация объектов, содержащих дескрипторы без специальной обработки, в лучшем случае вызовет ошибки, но вполне может вызвать и менее заметные проблемы. И даже если вам удастся наладить этот процесс, сериализация не имеет смысла для многих типов дескрипторов.

Таким образом, для сериализации обязательно согласие — только автор типа будет знать, будет копирование объекта поле за полем (что фактически делает сериализация) чем-то полезным. Вы можете согласиться на сериализацию, применив `SerializableAttribute` к своему классу. В отличие от большинства атрибутов, он обрабатывается особым образом в формате метаданных .NET — все кончается тем, что он устанавливает флаг в той части, где происходит определение класса. См. врезку «Атрибуты или пользовательские атрибуты?» для дополнительной информации.

Применяя атрибут `SerializableAttribute`, вы даете системе сериализации .NET разрешение брать непосредственно поля вашего класса и записывать

их значения в поток. Вы также даете разрешение обходить обычные конструкторы при восстановлении экземпляра вашего типа из сериализованного потока. (Можно предоставить специальный конструктор для целей сериализации, как я показал в главе 8, но если этого не сделать, сериализация приведет к появлению экземпляров вашего типа без вызова какого-либо конструктора вообще. Это одна из причин того, что сериализация — это функция CLR, а не библиотеки.) Вы можете отказаться от сериализации отдельных полей, применив атрибут `NonSerializedAttribute`.

Кстати, в библиотеках классов .NET есть несколько механизмов, которые выполняют работу, аналогичную сериализации CLR. На самом деле число опций несколько озадачивает, поскольку `XmlSerializer`, `DataContractSerializer`, `NetDataContractSerializer` и `DataContractJsonSerializer` предлагают различные форматы и философию сериализации. О наиболее используемых я расскажу в главе 15, а пока упоминаю их лишь потому, что они определяют многочисленные атрибуты. Однако поскольку все эти другие формы сериализации — это просто библиотечные функции, а не встроенные сервисы времени выполнения, их атрибуты не получают особой обработки от CLR.

JIT-компиляция

Есть несколько атрибутов, которые влияют на то, как JIT-компилятор генерирует код. Вы можете применить `MethodImplAttribute` к методу, передавая значения из перечисления `MethodImplOptions`. Его значение `NoInlining` гарантирует, что всякий раз, когда ваш метод вызывается другим методом, это будет полный вызов метода. Без него JIT-компилятор иногда будет просто копировать код метода прямо в вызывающий код.

В общем случае встраивание оставляют включенным. JIT-компилятор встраивает только небольшие методы, и это особенно важно для крошечных методов вроде доступа к свойствам. Для простых свойств, основанных на полях, для вызова методов доступа с помощью обычной функции часто требуется больше кода, чем при встраивании, поэтому такая оптимизация может привести к уменьшению и ускорению кода. (Даже если код не меньше, он все равно может быть быстрее, потому что вызовы функций могут быть на удивление затратными. Современные процессоры имеют тенденцию обрабатывать длинные последовательные потоки инструкций более эффективно, чем код, который скакает из одного места в другое.) Однако встраивание — это оптимизация с заметными побочными эффектами, так как встроенный метод не получает собственный кадр стека. Ранее я упо-

минал некоторые диагностические API, которые вы можете использовать для проверки стека, и встраивание изменит количество сообщаемых кадров стека. Если вы просто хотите задать вопрос: «Какой метод вызывает меня?», описанные выше атрибуты информации о вызывающей процедуре обеспечивают более эффективный способ узнать это и встраивание на них не подействует. Но если у вас есть код, который по какой-либо причине проверяет стек, встраивание вполне способно его запутать. Поэтому иногда бывает полезно его отключить.

И наоборот, вы можете использовать `AggressiveInlining`, который заставит JIT-компилятор встраивать методы, для которых в ином случае он бы использовал обычные вызовы. Если вы определили конкретный метод как высокочувствительный к производительности, возможно, стоит попробовать этот параметр, чтобы выяснить, окажет ли он какое-либо влияние. Хотя следует помнить, что он может сделать код как медленнее, так и быстрее — это будет зависеть от обстоятельств. И наоборот, вы можете отключить все оптимизации с помощью параметра `NoOptimization` (хотя документация подразумевает, что он предназначен больше для команды CLR в Microsoft, нежели для пользователей, потому что про него сказано, что он нужен для «отладки возможных проблем генерации кода»).

Другим атрибутом, влияющим на оптимизацию, является атрибут `DebuggableAttribute`. Компилятор C# автоматически применяет его к вашей сборке в `Debug`-сборках. Атрибут делает CLR менее агрессивной в отношении определенных оптимизаций, особенно тех, которые влияют на время жизни переменной, и тех, которые изменяют порядок выполнения кода. Обычно компилятор может изменять такие вещи, если конечный результат кода не меняется, но это может привести к путанице, если вклиниться в середину оптимизированного метода с помощью отладчика. Этот атрибут гарантирует, что отслеживать значения переменных и поток выполнения в этом сценарии будет достаточно легко.

STAThread и MTAThread

Приложения, которые работают только в Windows и представляют пользовательский интерфейс (например, любые, которые используют платформы .NET WPF или Windows Forms), обычно имеют атрибут `[STAThread]` в своем методе `Main` (хотя вы не всегда его видите, потому что точка входа для таких приложений часто генерируется системой сборки). Это инструкция для уровня взаимодействия COM в CLR, но ее применение более широкое: этот

атрибут необходим для `Main`, если вы хотите, чтобы ваш основной поток содержал элементы пользовательского интерфейса.

Различные функции пользовательского интерфейса Windows внутри полагаются на СОМ. Например, его использует буфер обмена, как и некоторые другие виды элементов управления. СОМ имеет несколько моделей потоков, и только одна из них совместима с потоками пользовательского интерфейса. Одна из основных причин этого заключается в том, что элементы пользовательского интерфейса привязаны к потокам. Поэтому СОМ должен убедиться, что та или иная работа выполняется им в нужном потоке. Кроме того, если поток пользовательского интерфейса не проверяет и не обрабатывает сообщения регулярно, это может привести ко взаимоблокировке. Если вы не сообщите СОМ, что конкретный поток является потоком пользовательского интерфейса, он опустит эти проверки и у вас могут возникнуть проблемы.



Даже если вы не пишете код пользовательского интерфейса, ряду сценариев взаимодействия необходим атрибут `[STAThread]`, поскольку некоторые компоненты СОМ не могут работать без него. Тем не менее работа с пользовательским интерфейсом является наиболее распространенной причиной для его использования.

Поскольку среда CLR управляет СОМ за вас, ей необходимо знать, когда нужно сообщить СОМ о том, что определенный поток должен обрабатываться как поток пользовательского интерфейса. Когда вы создаете новый поток явно, используя методы, показанные в главе 16, вы можете настроить его режим потока СОМ, но основной поток — это особый случай. CLR создает его за вас, когда ваше приложение запускается, и к тому времени, когда ваш код начинает выполняться, слишком поздно менять его настройки. Размещение атрибута `[STAThread]` в методе `Main` сообщает CLR, что ваш основной поток должен быть инициализирован для совместимого с пользовательским интерфейсом поведения СОМ.

STA — это сокращение от *single-threaded apartment* (однопотоковое подразделение). Потоки, участвующие в СОМ, всегда принадлежат либо к STA, либо к многопотковому подразделению (*multithreaded apartment*, MTA). Существуют и другие виды подразделений, но членство в них всегда временно; когда поток начинает использовать СОМ, он должен выбрать либо режим STA, либо режим MTA. Поэтому, вполне ожидаемо, существует также атрибут `[MTAThread]`.

Межпрограммное взаимодействие

Службы взаимодействия (interop) CLR определяют множество атрибутов. Большинство из них обрабатываются непосредственно CLR, потому что взаимодействие является неотъемлемой особенностью среды выполнения. Поскольку атрибуты имеют смысл только в контексте механизмов, которые они поддерживают, и поскольку их много, я не буду описывать их здесь полностью. Но листинг 14.16 показывает кое-что из того, что они могут делать.

Листинг 14.16. Атрибуты взаимодействия

```
[DllImport("advapi32.dll", CharSet = CharSet.Unicode, SetLastError = true,
           EntryPoint = "LookupPrivilegeValueW")]
internal static extern bool LookupPrivilegeValue(
    [MarshalAs(UnmanagedType.LPTStr)] string lpSystemName,
    [MarshalAs(UnmanagedType.LPTStr)] string lpName,
    out LUID lpLuid);
```

Здесь несколько более сложным способом используются два атрибута взаимодействия, которые мы видели ранее в листинге 14.6. Пример вызывает функцию, предоставляемую advapi32.dll, частью Win32 API. Первый аргумент атрибута `DllImport` говорит нам, что далее уровню взаимодействия будет предоставлена дополнительная информация (чего не было в более раннем примере). Этот API имеет дело со строками, поэтому взаимодействию необходимо знать, какое именно представление символов используется. В этом конкретном API используется распространенная идиома Win32: он возвращает логическое значение, указывающее на успешность или ошибку, но также использует API Windows под названием `SetLastError` для предоставления дополнительной информации в случае сбоя. Свойство `SetLastError` атрибута приказывает уровню взаимодействия получить это значение сразу после вызова этого API, чтобы .NET-код мог проверить его при необходимости. Свойство `EntryPoint` связано с тем фактом, что API Win32, принимающие строки, иногда бывают двух форм: для работы с 8-битными или 16-битными символами (для экономии памяти в Windows 95 поддерживался только 8-битный текст) и что мы хотим вызвать 16-битную форму (т. е. Wide, отсюда и суффикс `W`). Затем для двух строковых аргументов использован атрибут `MarshalAs`, который сообщает уровню взаимодействия, какое из множества различных строковых представлений, доступных в неуправляемом коде, ожидается этим конкретным API.

Определение и использование атрибутов

Подавляющее большинство атрибутов, с которыми вы столкнетесь, не относится к среде выполнения или компилятору. Они определяются библиотеками классов и работают, только если вы используете соответствующие библиотеки или платформы. В своем собственном коде вы можете делать то же самое — определять свои собственные типы атрибутов. Поскольку атрибуты ничего не делают сами по себе — они даже не создаются, если что-то не ожидает их увидеть, — обычно определение типа атрибута полезно, только если вы пишете какую-то платформу, особенно управляемую отражением.

Например, платформы юнит-тестов часто находят написанные вами классы тестов с помощью отражения и позволяют вам контролировать поведение системы тестирования с помощью атрибутов. Другой пример — это то, как Visual Studio использует отражение для обнаружения свойств редактируемых объектов при дизайне (например, элементов управления пользовательского интерфейса) и ищет определенные атрибуты, которые позволяют настраивать поведение редактирования. Другое применение атрибутов — это настройка исключений для правил, применяемых инструментами статического анализа кода Visual Studio, когда вы помечаете свой код соответствующими атрибутами. Во всех этих случаях какой-либо инструмент или среда проверяет ваш код и решает, что делать, основываясь на том, что обнаруживает. Атрибуты очень хорошо вписываются в такой сценарий.

Например, атрибуты могут быть полезны, если вы пишете приложение, которое разрешено расширять конечным пользователям. Вы можете поддерживать загрузку внешних сборок, которые дополняют поведение вашего приложения, — такое часто называют моделью подключаемых модулей. Может оказаться полезным атрибут, который позволяет такому модулю предоставлять о себе описательную информацию. Атрибуты использовать не обязательно — скорее всего, вы определите хотя бы один интерфейс, который требуется реализовать всем модулям, и в этом интерфейсе могут быть члены для получения необходимой информации. Однако одним из преимуществ использования атрибутов является то, что вам не нужно создавать экземпляр модуля только для получения информации описания. Это позволяет предоставить пользователю подробности касательно модуля перед его фактической загрузкой, что может оказаться важным в том случае, если создание модуля имеет побочные эффекты, которые могут не устраивать пользователя.

Типы атрибутов

Листинг 14.17 показывает, как может выглядеть атрибут, содержащий информацию о подключаемом модуле.

Листинг 14.17. Тип атрибута

```
[AttributeUsage(AttributeTargets.Class)]
public class PluginInformationAttribute : Attribute
{
    public PluginInformationAttribute(string name, string author)
    {
        Name = name;
        Author = author;
    }

    public string Name { get; }

    public string Author { get; }

    public string Description { get; set; }
}
```

Чтобы действовать в качестве атрибута, тип должен быть производным от базового класса `Attribute`. Хотя `Attribute` определяет различные статические методы для обнаружения и извлечения атрибутов, он не представляет особого интереса для самих экземпляров. Мы не наследуем из него какой-либо конкретный функционал; мы поступаем так лишь потому, что компилятор не позволит использовать тип в качестве атрибута, если он не является производным от `Attribute`.

Обратите внимание, что имя моего типа заканчивается словом `Attribute`. Это не непреложное требование, но чрезвычайно широко распространенное соглашение. Как вы уже видели, оно даже встроено в компилятор, который автоматически добавляет суффикс `Attribute`, если вы пропускаете его при применении атрибута. Поэтому обычно нет причин, чтобы не следовать этому соглашению.

Я аннотировал свой тип атрибута атрибутом. Большинство типов атрибутов аннотируются атрибутом `AttributeUsageAttribute` с указанием целей, к которым атрибут может быть осмысленно применен. Компилятор C# заставляет так поступать. Поскольку мой атрибут в листинге 14.17 говорит, что может быть применен только к классам, компилятор выдаст ошибку, если попытаться применить его к чему-либо еще.



Как вы видели, иногда при применении атрибута нужно указать его цель. Например, когда атрибут появляется перед методом, его целью является метод, если только вы не квалифицируете его с помощью префикса `return:`. Возможно, вы надеялись, что сможете отбросить эти префиксы при использовании атрибутов, которые предназначены только для определенных членов. Например, если атрибут может быть применен только к сборке, действительно ли вам нужен квалификатор `assembly:?` Но C# не позволит вам его отбросить. Он использует `AttributeUsageAttribute` только для проверки того, что атрибут не был применен неправильно.

Атрибут определяет только один конструктор, поэтому любой код, который его использует, должен будет передавать необходимые конструктору аргументы, как это делается в листинге 14.18.

Листинг 14.18. Применение атрибута

```
[PluginInformation("Reporting", "Endjin Ltd.")]
public class ReportingPlugin
{
    ...
}
```

Классы атрибутов могут без ограничений определять множественные перегрузки конструктора для поддержки различных наборов информации. Они также могут определять свойства как способ поддержки необязательных фрагментов информации. Мой атрибут определяет свойство `Description`, которое не является обязательным, поскольку конструктор не требует для него значения, но которое я могу установить, используя синтаксис, который я описал ранее в этой главе. Листинг 14.19 показывает, как это выглядит в случае моего атрибута.

Листинг 14.19. Предоставление необязательного значения свойства для атрибута

```
[PluginInformation("Reporting", "Endjin Ltd.",
    Description = "Automated report generation")]
public class ReportingPlugin
{
    ...
}
```

Пока что ничего из того, что я показал, не приведет к созданию экземпляра моего типа `PluginInformationAttribute`. Эти аннотации являются просто

инструкциями для того, как атрибут должен быть инициализирован, если он кому-то потребуется. Итак, чтобы от атрибута была польза, мне нужно написать код, который будет его проверять.

Извлечение атрибутов

Вы можете узнать, был ли применен определенный тип атрибута, используя API отражения, который также может создать для вас экземпляр атрибута. В главе 13 я показал все типы отражений, представляющие собой различные цели, к которым могут применяться атрибуты, такие как `MethodInfo`, `Type` и `PropertyInfo`. Все они реализуют интерфейс `ICustomAttributeProvider`, который показан в листинге 14.20.

Листинг 14.20. `ICustomAttributeProvider`

```
public interface ICustomAttributeProvider
{
    object[] GetCustomAttributes(bool inherit);
    object[] GetCustomAttributes(Type attributeType, bool inherit);
    bool IsDefined(Type attributeType, bool inherit);
}
```

Метод `IsDefined` просто сообщает вам, присутствует ли определенный тип атрибута, но не создает его. Две перегрузки `GetCustomAttributes` создают атрибуты и возвращают их. (Это точка, в которой создаются атрибуты. Кроме того, это происходит когда устанавливаются какие-либо свойства, заданные в аннотациях.) Первая перегрузка возвращает все атрибуты, примененные к цели, а вторая позволяет запрашивать только атрибуты определенного типа.

Все эти методы принимают аргумент `bool`, который позволяет указать, нужны вам атрибуты, примененные только непосредственно к проверяемой цели, или также атрибуты, определенные базовым типом или типами.

Этот интерфейс был введен в .NET 1.0, поэтому он не использует обобщений, что означает, что придется приводить возвращаемые объекты. К счастью, статический класс `CustomAttributeExtensions` определяет несколько методов расширения. Вместо определения их для интерфейса `ICustomAttributeProvider` он расширяет классы отражения, которые предлагаю атрибуты. Например, если у вас есть переменная типа `Type`, вы мо-

жете вызвать для него `GetCustomAttribute<PluginInformationAttribute>()`, который будет создавать и возвращать атрибут информации о плагине, или `null`, если атрибут отсутствует. Листинг 14.21 использует эту технику, чтобы показать всю информацию о модулях из всех библиотек DLL в определенной папке.

Листинг 14.21. Отображение информации о подключаемых модулях

```
static void ShowPluginInformation(string pluginFolder)
{
    var dir = new DirectoryInfo(pluginFolder);
    foreach (var file in dir.GetFiles("*.dll"))
    {
        Assembly pluginAssembly = Assembly.LoadFrom(file.FullName);
        var plugins =
            from type in pluginAssembly.ExportedTypes
            let info = type.GetCustomAttribute<PluginInformationAttribute>()
            where info != null
            select new { type, info };
        foreach (var plugin in plugins)
        {
            Console.WriteLine($"Plugin type: {plugin.type.Name}");
            Console.WriteLine(
                $"Name: {plugin.info.Name}, written by
                {plugin.info.Author}");
            Console.WriteLine($"Description: {plugin.info.Description}");
        }
    }
}
```

С этим может возникнуть потенциальная проблема. Я сказал, что одним из преимуществ атрибутов является то, что они могут быть получены без создания экземпляров их целевых типов. В данном случае это так — я не создаю никаких подключаемых модулей в листинге 14.21. Однако я загружаю их сборки, и возможным побочным эффектом перечисления модулей будет запуск статических конструкторов в их DLL. Поэтому хотя я не выполняю какой-либо код из этих DLL намеренно, я не могу гарантировать, что никакой код из них не будет запущен. Если моя цель — предоставить пользователю список модулей, после чего загрузить и запустить только те из них, которые были выбраны явно, мое решение неудачно, потому что я дал коду модулей возможность запуститься. Тем не менее можно это исправить.

Загрузка только для отражения

Не нужно загружать сборку полностью, чтобы получить информацию об атрибутах. Как я уже говорил в главе 13, вы можете загрузить сборку только для целей отражения. Это предотвращает запуск любого кода в сборке, но позволяет проверять содержащиеся в нем типы. Тем не менее это представляет собой проблему в случае атрибутов. Обычный способ проверить свойства атрибута — создать его экземпляр, вызвав `GetCustomAttributes` или связанный метод расширения. Поскольку это включает в себя создание атрибута (что означает выполнение некоторого кода), он не поддерживается для сборок, загруженных для отражения (даже если рассматриваемый тип атрибута был определен в другой сборке, которая была полностью загружена обычным способом). Если я изменю листинг 14.21, чтобы загрузить сборку с помощью `ReflectionOnlyLoadFrom`, то вызов `GetCustomAttribute<PluginInformationAttribute>` вызовет исключение.

При загрузке только для отражения необходимо использовать метод `GetCustomAttributesData`. Вместо того чтобы создавать атрибут, он возвращает информацию, хранящуюся в метаданных, — инструкции по созданию атрибута. В листинге 14.22 показана версия соответствующего кода из листинга 14.21, модифицированная для работы таким образом.

Листинг 14.22. Получение атрибутов с помощью контекста только для отражения

```
Assembly pluginAssembly = Assembly.ReflectionOnlyLoadFrom(
    file.FullName);

var plugins =
    from type in pluginAssembly.ExportedTypes
    let info = type.GetCustomAttributesData().SingleOrDefault(
        attrData => attrData.AttributeType.FullName ==
            pluginAttributeType.FullName)
    where info != null
    let description = info.NamedArguments
        .SingleOrDefault(a => a.MemberName == "Description")

    select new
    {
        type,
        Name = (string) info.ConstructorArguments[0].Value,
        Author = (string) info.ConstructorArguments[1].Value,
        Description =
            description == null ? null : description.TypedValue.Value
    };
}
```

```
foreach (var plugin in plugins)
{
    Console.WriteLine($"Plugin type: {plugin.type.Name}");
    Console.WriteLine($"Name: {plugin.Name}, written by {plugin.Author}");
    Console.WriteLine($"Description: {plugin.Description}");
}
```

Код довольно громоздкий, потому что мы не возвращаем экземпляр атрибута. `GetCustomAttributesData` возвращает коллекцию объектов `CustomAttributeData`. Листинг 14.22 использует оператор LINQ `SingleOrDefault`, чтобы найти запись для `PluginInformationAttribute`, и, если она присутствует, переменная `info` в запросе будет содержать ссылку на соответствующий объект `CustomAttributeData`. Затем код выбирает аргументы конструктора и значения свойств, используя свойства `ConstructorArguments` и `NamedArguments`, что позволяет ему получить три описательных текстовых значения, встроенных в атрибут.

Пример показывает и то, что контекст только для отражения добавляет сложности, поэтому его следует использовать, только если требуются преимущества, которые он предлагает. Одним из преимуществ является то, что он не будет запускать ни одну из загруженных вами сборок. Он также может загружать сборки, которые могут быть отклонены при обычной загрузке (например, потому что они предназначены для конкретной архитектуры процессора, которая не соответствует вашему процессору). Но если вам не нужна опция «только отражение», более прямой доступ к атрибутам, показанный в листинге 14.21, подойдет лучше.

Итог

Атрибуты обеспечивают способ встраивания пользовательских данных в метаданные сборки. Вы можете применять атрибуты к типу, любому члену типа, параметру, возвращаемому значению или даже целой сборке или одному из ее модулей. Несколько атрибутов обрабатываются CLR особым образом и несколько управляют функциями компилятора, но большинство из них не имеют встроенного поведения, действуя просто как пассивные информационные контейнеры. Атрибуты даже не создаются, если что-то не пытается проверить их наличие. Все это делает атрибуты наиболее полезными в системах с управляемым отражением поведением — если у вас уже есть один из объектов API отражения, например `ParameterInfo` или `Type`,

вы можете напрямую запросить его атрибуты. Поэтому чаще всего можно наблюдать использование атрибутов в средах, которые проверяют ваш код с помощью отражения, например в модульных тестах, средах сериализации, управляемых данными элементах пользовательского интерфейса, таких как панель свойств Visual Studio, или в библиотеках загружаемых модулей. Если вы используете подобную среду, то вы, как правило, имеете возможность регулировать ее поведение, помечая свой код атрибутами, которые она распознает. Если вы сами пишете что-то подобное, то имеет смысл определить собственные типы атрибутов.

Файлы и потоки

Большинство техник, которые я до сих пор демонстрировал в этой книге, касаются информации, которая содержится в объектах и переменных. Такой тип информации хранится в памяти конкретного процесса, но, чтобы быть полезной, программе необходимо взаимодействовать с миром и за пределами процесса. Этого можно добиться с помощью UI-фреймворков, но есть одна конкретная абстракция, которую можно использовать для многих видов взаимодействия с внешним миром: *поток*.

Потоки настолько широко используются в вычислениях, что вы, без сомнения, уже знакомы с ними, и поток .NET во многом напоминает потоки в большинстве других систем программирования: это просто последовательность байтов. Все это делает поток полезной абстракцией для множества часто встречающихся функций, таких как файл на диске или тело ответа HTTP. Консольное приложение использует потоки для представления своих ввода и вывода. Если вы запускаете такую программу в интерактивном режиме, текст, который пользователь вводит на клавиатуре, становится входным потоком программы, а все, что программа записывает в свой выходной поток, появляется на экране. Программа не обязательно в курсе, какой у нее тип ввода или вывода — вы можете перенаправлять эти потоки с помощью консольных программ. Например, входной поток может фактически предоставлять содержимое файла на диске или же это может быть вывод какой-либо другой программы.



Не все API ввода-вывода являются потоковыми. Например, в дополнение к потоку ввода класс `Console` предоставляет метод `ReadKey`, который дает информацию о том, какая именно клавиша была нажата. Метод работает только в том случае, если ввод осуществляется с клавиатуры. Иными словами, хотя вы и можете писать программы, которые не заботят, поступает ли их ввод интерактивно или из файла, некоторые программы более избирательны.

Потоковые API предоставляют вам необработанные байтовые данные. Однако можно работать на другом уровне. Например, существуют текстово ориентированные API, которые могут оберачивать лежащие в основе потоки, поэтому вы можете работать с символами или строками вместо необработанных байтов. Существуют также различные механизмы *серIALIZации*, которые позволяют преобразовывать объекты .NET в потоковое представление, которое позже можно превратить в объекты. Это позволяет сохранять состояние объекта или отправлять это состояние по сети. Позже я покажу эти высокоуровневые API, но сначала давайте взглянем на саму абстракцию потока.

Класс Stream

Класс Stream определяется в пространстве имен `System.IO`. Это абстрактный базовый класс с конкретными производными типами, такими как `FileStream` или `GZipStream`, представляющими определенные виды потоков. В листинге 15.1 показаны три наиболее важных члена класса `Stream`. У него есть несколько других членов, но эти — сердце абстракции. (Как вы увидите позже, есть также асинхронные версии `Read` и `Write`. .NET Core 3.0 и .NET Standard также добавляют перегрузки, которые заменяют массив одним из типов *диапазона*, описанных в главе 18. Все, что я говорю в этом разделе об этих методах, также относится к асинхронным и основанным на диапазоне формам.)

Листинг 15.1. Самые важные члены Stream

```
public abstract int Read(byte[] buffer, int offset, int count);
public abstract void Write(byte[] buffer, int offset, int count);
public abstract long Position { get; set; }
```

Некоторые потоки доступны только для чтения. Например, когда поток ввода для консольного приложения представляет клавиатуру или вывод какой-либо другой программы, для программы нет никакого реального способа записи в этот поток. (В целях согласованности, даже если вы используете перенаправление ввода для запуска консольного приложения с файлом в качестве входных данных, входной поток будет только для чтения.) Некоторые потоки доступны только для записи, например выходной поток консольного приложения. Если вы вызываете `Read` для потока только для записи или `Write` для потока только для чтения, возникнет исключение `NotSupportedException`.



Класс `Stream` определяет различные свойства типа `bool`, которые описывают возможности потока, поэтому вам не нужно ждать, пока вы получите исключение. Вы можете проверить свойства `CanRead` или `CanWrite`.

И `Read`, и `Write` принимают массив `byte[]` в качестве первого аргумента и копируют данные в него или из него соответственно. Последующие аргументы `offset` и `count` указывают элемент массива, с которого нужно начинать, и количество байтов для чтения или записи; вам не обязательно использовать весь массив. Обратите внимание, что вы не можете указать смещение для чтения или записи внутри потока. Это управляется свойством `Position` — работа начинается с нуля, но каждый раз, когда вы читаете или записываете, позиция увеличивается на количество обработанных байтов.

Обратите внимание, что метод `Read` возвращает `int`. Это число говорит вам, сколько байтов было прочитано из потока — метод не гарантирует представление запрошенного вами объема данных. Одна из очевидных причин этого заключается в том, что вы можете достичь конца потока, поэтому, даже если вы попросили прочитать 100 байт и записать их в ваш массив, между текущей позицией и концом потока может оставаться только 30 байт данных. Тем не менее это не единственная причина, по которой вы можете получить меньше данных, чем просили, и это часто застает врасплох. Поэтому ради тех, кто решил быстренько ознакомиться с данной темой, я дам эту информацию в виде грозного предупреждения.



Если вы запрашиваете более одного байта за раз, `Stream` всегда может вернуть меньше данных, чем вы запрашивали в `Read`. Никогда не полагайтесь на то, что вызов `Read` вернет столько данных, сколько запрошено, даже если у вас есть веские основания думать, что эти данные доступны.

Причина, по которой `Read` не так прост, заключается в том, что некоторые потоки — это потоки в реальном времени и они представляют собой источник информации, который генерирует данные постепенно по мере выполнения программы. Например, если консольное приложение работает в интерактивном режиме, его входной поток может предоставлять данные только с той скоростью, с которой их вводит пользователь; поток, представляющий данные, принимаемые по сетевому соединению, может предоставлять дан-

ные только с той скоростью, с которой они поступают. Если вы вызываете `Read` и запрашиваете больше данных, чем доступно в данный момент, поток может подождать, пока у него будет столько, сколько вы просили, но не обязательно — он может вернуть то количество данных, которые он имеет на момент вызова. (Единственная ситуация, в которой метод обязан ждать перед возвратом, это если у него вообще нет данных, но он еще не достиг конца потока. Он должен вернуть как минимум один байт, поскольку возвращаемое значение `0` указывает на конец потока.) Если вы хотите убедиться, что прочитали определенное количество байтов, вам нужно проверить, вернул ли `Read` меньше, чем требовалось, и, если необходимо, продолжить вызовы, пока не получите нужное количество информации. В листинге 15.2 показано, как это сделать.

Листинг 15.2. Чтение определенного количества байтов

```
static int ReadAll(Stream s, byte[] buffer, int offset, int length)
{
    if ((offset + length) > buffer.Length)
    {
        throw new ArgumentException(
            "Buffer too small to hold requested data");
    }

    int bytesReadSoFar = 0;
    while (bytesReadSoFar < length)
    {
        int bytes = s.Read(
            buffer, offset + bytesReadSoFar, length - bytesReadSoFar);
        if (bytes == 0)
        {
            break;
        }
        bytesReadSoFar += bytes;
    }

    return bytesReadSoFar;
}
```

Обратите внимание, что для обнаружения конца потока код проверяет возвращаемое значение `Read` на содержание значения `0`. Без этого в случае достижения конца потока перед тем, как прочитать запрошенное количество данных, он будет работать бесконечно. Это означает, что, достигнув конца потока, этот метод вынужден будет предоставить меньше данных, чем за-

прашивала вызывающая сторона, и может показаться, что проблема на самом деле не решена. Однако это исключает ситуацию, когда вы получаете меньше, чем просили, несмотря на то что еще не достигли конца потока. (Конечно, вы можете изменить метод так, чтобы он генерировал исключение, если достигнет конца потока, прежде чем предоставить указанное количество байт. Таким образом, если метод вообще возвращает результат, он гарантированно вернет ровно столько байтов, сколько было запрошено.)

`Stream` предлагает более простой способ чтения. Метод `ReadByte` возвращает один байт, если только вы не достигли конца потока, в момент чего он возвращает значение `-1`. (Тип возвращаемого значения — `int`, что позволяет ему возвращать любое возможное значение `byte`, а также отрицательные значения.) Это позволяет избежать проблемы, связанной с возвратом только части запрошенных данных, потому что, если вы вообще что-то получите, это будет ровно одному байту. Но это не особенно удобно и быстро, если вы хотите читать большие фрагменты данных.

Метод `Write` не имеет ни одной из этих проблем. Если он срабатывает, то всегда принимает все предоставленные данные. Конечно, может произойти сбой — из-за ошибки метод может вызвать исключение прежде, чем ему удастся записать все данные (например, из-за нехватки места на диске или потери сетевого подключения).

Позиция и поиск

Потоки автоматически обновляют свою текущую позицию каждый раз, когда вы читаете или записываете. Как видно в листинге 15.1, можно установить свойство `Position` и попытаться перейти непосредственно к определенной позиции. Нет гарантии, что это сработает, потому что поддержка позиционирования не всегда возможна. Например, поток, который представляет данные, принятые по сетевому соединению TCP, может производить данные неограниченное время — пока соединение остается открытым, а другой конец продолжает отправлять данные, поток будет продолжать обрабатывать вызовы `Read`. Соединение может оставаться открытым в течение многих дней и получить за это время терабайты данных. Если такой поток позволяет вам установить свойство `Position`, позволяя вашему коду возвращаться назад и перечитывать полученные ранее данные, то ему понадобится найти место для хранения каждого полученного байта на тот случай, если код, использующий поток, захочет к нему вернуться. Поскольку для

этого может потребоваться хранить больше данных, чем позволяет место на диске, это явно непрактично. Поэтому некоторые потоки будут выдавать исключение `NotSupportedException` при попытке установить свойство `Position`. (Существует свойство `CanSeek`, которое вы можете использовать для проверки того, поддерживает ли конкретный поток изменение позиции. Благодаря этому, как и в случае потоков, доступных только для чтения и только для записи, вам не нужно ждать, пока вы получите исключение, чтобы выяснить, сработает ли установка позиции.)

Помимо свойства `Position` `Stream` также определяет метод `Seek`, сигнатура которого показана в листинге 15.3. Метод позволяет вам указать требуемую позицию относительно текущей позиции потока. (Он также приведет к исключению `NotSupportedException` для потоков, которые не поддерживают поиск.)

Листинг 15.3. Метод `Seek`

```
public abstract long Seek(long offset, SeekOrigin origin);
```

Если вы передадите `SeekOrigin.Current` в качестве второго аргумента, он установит позицию, добавив первый аргумент к текущей позиции. Можно передать отрицательное смещение, если требуется сместиться назад. Вы также можете передать `SeekOrigin.End`, чтобы установить позицию равной указанному количеству байтов от конца потока. Передача `SeekOrigin.Begin` имеет тот же логический эффект, что и непосредственная установка `Position`, — устанавливает позицию относительно начала потока¹.

Сброс

Как и в случае потоковых API в других системах программирования, запись данных в поток не обязательно приводит к немедленному достижению данными точки назначения. Когда вызов `Write` возвращается, вам известно лишь то, что он куда-то скопировал ваши данные; но это может быть не конечная точка, а буфер в памяти. Например, если вы записываете один байт в поток, представляющий файл на диске, объект потока обычно откладывает

¹ Хотя `Seek` логически эквивалентен установке `Position`, некоторые потоки без видимых причин обрабатывают их немного по-разному. `BufferedStream` библиотеки классов при установке `Position` отбрасывает все ранее прочитанные данные, тогда как метод `Seek` проверяет, не указывает ли новая позиция на данные, которые уже загружены.

фактическую запись до тех пор, пока у него не накопится достаточно данных, чтобы запись стоила усилий. Диски — это устройства на основе блоков. Это означает, что запись происходит в виде фрагментов фиксированного размера, обычно размером в несколько килобайтов. Поэтому, прежде чем записывать, обычно имеет смысл подождать, пока не скопится достаточно данных для заполнения блока.

Эта буферизация обычно полезна: она повышает производительность записи и позволяет игнорировать детали работы диска. Однако недостатком является то, что, если вы записываете данные только изредка (например, при записи сообщений об ошибках в лог-файл), вы легко можете столкнуться с длительными задержками между записью в поток и попаданием данных на диск. Это может вызывать затруднения у того, кто пытается диагностировать проблему, просматривая файлы журналов работающей в данный момент программы. И что еще более ужасно: если в вашей программе произойдет сбой, все, что находится в буферах потока и еще не попало на диск, скорее всего, будет потеряно.

Поэтому класс `Stream` содержит метод `Flush`. Он позволяет сообщить потоку, что вы хотите, чтобы он выполнил все необходимое для обеспечения того, чтобы любые буферизованные данные были записаны куда нужно, даже если это означает неоптимальное использование буфера.



При использовании `FileStream` метод `Flush` не обязательно гарантирует, что сбрасываемые данные уже попали на диск. Он просто заставляет поток передать данные ОС. До вашего вызова `Flush` ОС вообще не знала об этих данных, поэтому, если процесс внезапно завершится, данные будут потеряны. После возврата `Flush` ОС получила все, что записал ваш код, поэтому процесс может быть остановлен без потери данных. Тем не менее ОС может самостоятельно выполнять дополнительную буферизацию, поэтому, если происходит сбой питания до того, как ОС приступит к записи на диск, данные все равно будут потеряны. Если вам нужна постоянная гарантия того, что данные были записаны (а не просто гарантия передачи их ОС), можно использовать флаг `WriteThrough`, описанный в разделе «Класс `FileStream`» на с. 779, или вызвать перегрузку `Flush`, которая принимает `bool` для принудительного сброса на диск.

Поток автоматически сбрасывает свое содержимое при вызове `Dispose`. Следует использовать `Flush` только тогда, когда хотите сохранить поток от-

крытым после записи буферизованных данных. Это особенно важно в случае продолжительных периодов, когда поток открыт, но неактивен. (Если поток представляет собой сетевое соединение и если ваше приложение зависит от быстрой передачи данных, например в случае приложения онлайн-чата или игры, стоит вызывать `Flush`, даже если вы ожидаете лишь довольно короткий период бездействия.)

Копирование

Иногда может пригодиться копирование всех данных из одного потока в другой. Для этого не составит труда написать цикл, но это необязательно, потому что метод `CopyTo` класса `Stream` (или эквивалентный `CopyToAsync`) сделает все за вас. О нем особенно нечего говорить. Основная причина, по которой я о нем упоминаю, заключается в том, что разработчики нередко пишут свои собственные версии этого метода, так как не знают, что эта функциональность уже встроена в `Stream`.

Length

Некоторые потоки могут сообщать свою длину через свойство с предсказуемым именем `Length`. Как и в случае `Position`, тип этого свойства — `long`. Класс `Stream` использует 64-битные числа, поскольку потоки часто должны быть больше 2 ГБ, что было бы верхним пределом, если бы размеры и позиции были представлены `int`.

`Stream` также определяет метод `SetLength`, который позволяет вам задавать длину потока (если это поддерживается). Это можно использовать при записи большого количества данных в файл, чтобы обеспечить достаточно места для хранения всех необходимых данных, — лучше получить `IOException` сразу, нежели тратить время на операцию, которая обречена на провал и потенциально может вызвать проблемы в системе из-за использования всего свободного пространства. Однако многие файловые системы поддерживают разреженные файлы, что позволяет создавать файлы, размер которых намного превышает доступное свободное пространство, поэтому на практике вы можете не увидеть никаких ошибок, пока не начнете записывать ненулевые данные. Тем не менее если вы укажете длину, которая больше, чем поддерживает файловая система, `SetLength` вызовет исключение `ArgumentException`.

Не все потоки поддерживают операции длины. В документации по классу `Stream` указано, что свойство `Length` доступно только для потоков, поддерживающих `CanSeek`. Это связано с тем, что потоки, поддерживающие поиск, обычно являются потоками, в которых все содержимое потока известно и доступно заранее. Поиск недоступен для потоков, в которых содержимое создается во время выполнения (например, входные потоки, представляющие пользовательский ввод, или потоки, представляющие данные, принимаемые по сети), а длина очень часто не известна заранее. Что касается `SetLength`, то в документации говорится, что метод поддерживается только в потоках, которые поддерживают как запись, так и поиск. (Как и все члены, представляющие необязательные функции, `Length` и `SetLength` будут генерировать исключение `NotSupportedException`, если вы попытаетесь использовать их в потоках, которые их не поддерживают.)

Очистка

Некоторые потоки представляют ресурсы, внешние по отношению к среде выполнения .NET. Например, `FileStream` обеспечивает потоковый доступ к содержимому файла, поэтому ему необходимо получить от ОС дескриптор файла. Важно закрыть дескрипторы, когда вы закончите с ними работать; в противном случае вы можете помешать другим приложениям использовать файл. Следовательно, класс `Stream` реализует интерфейс `IDisposable` (описанный в главе 7), дающий возможность понять, когда это необходимо сделать. И как я упоминал ранее, потоки буферизации, такие как `FileStream`, сбрасывают свои буферы при вызове `Dispose` перед закрытием дескрипторов.

Не все типы потоков полагаются на вызов `Dispose`: `MemoryStream` работает полностью в памяти, поэтому о нем может позаботиться сборщик мусора. Но в общем случае, если вы вызвали создание потока, вы же должны и вызвать `Dispose`, когда поток вам больше не нужен.



В некоторых ситуациях поток будет вам предоставлен, но распоряжаться им — не ваша задача. Например, ASP.NET Core может предоставлять потоки для представления данных HTTP-запросов и ответов на них. Он создает их для вас и удаляет после того, как вы их использовали, поэтому вам не следует вызывать их методы `Dispose` самостоятельно.

По непонятной причине у класса `Stream` также есть метод `Close`. Но так сложилось исторически. Первая публичная бета-версия .NET 1.0 не определяла `IDisposable`, а в C# не было операторов `using` — ключевое слово использовалось только для директив `using`, которые вводят пространства имен в область видимости. Класс `Stream` нуждался в способе узнать, когда нужно очищать свои ресурсы, и, поскольку стандартного способа сделать это пока еще не было, в нем появилась собственная идиома. Он определил метод `Close`, который соответствовал терминологии, используемой во многих потоковых API в других системах программирования. `IDisposable` был добавлен до финальной версии .NET 1.0, и в класс `Stream` добавилась поддержка этого интерфейса, но остался и метод `Close`; его удаление свело бы на нет труд многих ранних разработчиков, использовавших бета-версии. Но метод `Close` является избыточным, и документация активно советует не использовать его. В ней сказано, что вместо этого следует вызывать `Dispose` (через оператор `using`). В вызове оператора `Close` нет ничего плохого, так как нет практической разницы между ним и `Dispose`. Однако `Dispose` является более распространенной идиомой и поэтому предпочтительнее.

Асинхронные операции

Класс `Stream` предлагает асинхронные версии `Read` и `Write`. Стоит помнить, что у них есть две формы. `Stream` впервые появился в .NET 1.0, поэтому он поддерживал то, что на тот момент было стандартным асинхронным механизмом, а именно модель асинхронного программирования (*Asynchronous Programming Model*, APM, описанную в главе 16), с использованием методов `BeginRead`, `EndRead`, `BeginWrite` и `EndWrite`. В настоящее время эта модель устарела и была заменена более новым асинхронным шаблоном на основе задач (*Task-based Asynchronous Pattern*, TAP, также описанным в главе 16). `Stream` поддерживает его с помощью методов `ReadAsync` и `WriteAsync`. Есть еще две операции, которые изначально не имели асинхронной формы, но теперь поддерживают TAP: `FlushAsync` и `CopyToAsync`. (Они поддерживают только TAP, поскольку к тому времени, когда Microsoft добавила эти методы, APM уже устарела.)



Избегайте старых, основанных на APM, форм `Begin/End` методов `Read` и `Write`. Их вообще не было ни в ранних версиях .NET Core, ни в .NET Standard до версии 2.0. Они появились в целях упрощения миграции существующего кода из .NET Framework в .NET Core, поэтому они поддерживаются только для устаревших сценариев.

Некоторые типы потоков реализуют асинхронные операции, используя крайне эффективные методы, которые напрямую соответствуют асинхронным возможностям базовой ОС.

(Так работает `FileStream`, а также различные потоки, которые .NET может предоставить для представления содержимого сетевых подключений.) Можно найти библиотеки с пользовательскими типами потоков, которые этого не делают, но даже тогда асинхронные методы останутся доступны, потому что базовый класс `Stream` способен использовать многопоточные методы.

При использовании асинхронных операций чтения и записи следует соблюдать осторожность: у потока есть только одно свойство `Position`. Чтение и запись зависят от текущего значения `Position`, а также обновляют его по окончании, поэтому в целом вы должны избегать запуска новой операции до завершения предыдущей. (Если вы хотите выполнить несколько одновременных операций чтения или записи из определенного файла, вы можете создать несколько потоковых объектов для этого файла или открыть файл в асинхронном режиме. `FileStream` имеет специальную обработку асинхронного доступа к файлам. Операции используют значение `Position` на момент начала, и после начала асинхронного чтения или записи вы можете изменить `Position` и начать другую операцию, не дожидаясь завершения всех предыдущих. Но это относится только к `FileStream` и только тогда, когда файл был открыт в асинхронном режиме.)

В .NET Core 3.0 и .NET Standard 2.1 добавился `IAsyncDisposable`, асинхронная форма `Dispose`. Его использует класс `Stream`, потому что удаление часто включает очистку, а это потенциально медленная операция.

Конкретные типы потоков

Класс `Stream` является абстрактным, поэтому для использования потока вам понадобится конкретный производный тип. В некоторых ситуациях его вам предоставят — в частности, веб-платформа ASP.NET Core предоставляет потоковые объекты, например для тела HTTP-запросов и ответов. Что-то подобное будет делать и клиентский класс `HttpClient`. Но иногда вам может понадобиться собственный объект потока. В этом разделе описываются некоторые из наиболее часто используемых типов, производных от `Stream`.

Класс `FileStream` представляет файл в файловой системе. Его я опишу в разделе «Файлы и каталоги» на с. 779.

`MemoryStream` позволяет вам создавать поток поверх массива `byte[]`. Вы можете либо взять существующий `byte[]` и обернуть его в `MemoryStream`, либо создать `MemoryStream` и затем заполнить его данными, вызвав `Write` (или асинхронный эквивалент). Закончив, вы можете получить заполненный `byte[]`, вызвав `ToByteArray` или `GetBuffer`. (`ToByteArray` выделяет новый массив с размером, основанным на количестве фактически записанных байтов. `GetBuffer` более эффективен, потому что он возвращает базовый массив, который использует `MemoryStream`. Но если только в результате записи не произойдет его полное заполнение, возвращаемый массив будет содержать больше байтов, чем было записано.) Класс может пригодиться, когда вы работаете с API, для которых требуется поток, и у вас его по какой-то причине нет. Например, большинство API сериализации, описанных далее в этой главе, работают с потоками, но вам может понадобиться использовать их в сочетании с каким-то другим API, который работает на основе `byte[]`. `MemoryStream` дает вам возможность соединить эти два представления.

И Windows, и Unix определяют механизм межпроцессного взаимодействия (IPC), позволяющий соединять два процесса через поток. Windows называет это именованными каналами. Unix также содержит механизм с таким именем, но он совершенно другой; однако там тоже имеется механизм, аналогичный именованным каналам Windows: доменные сокеты. Хотя в деталях именованные каналы Windows и доменные сокеты Unix различаются, в .NET для обоих общую абстракцию представляют различные классы, производные от `PipeStream`.

`BufferedStream` наследуется от `Stream`, но также принимает `Stream` в качестве параметра конструктора. Он добавляет слой буферизации, который нужен, если вам требуется выполнять небольшие операции чтения или записи в потоке, который предназначен для работы с операциями большего размера. (Вам не нужно использовать его при работе с `FileStream`, потому что он имеет собственный встроенный механизм буферизации.)

Существуют различные типы потоков, которые определенным образом преобразуют содержимое других потоков. Например, `DeflateStream`, `GZipStream` и `BrotliStream` реализуют три широко используемых алгоритма сжатия. Вы можете обернуть их вокруг других потоков, чтобы сжать данные, записанные в базовый поток, или распаковать прочитанные из него данные. (Они просто предоставляют службу сжатия самого низкого уровня. Если вы хотите работать с популярным форматом ZIP для пакетов сжатых файлов, используйте класс `ZipArchive`.) Существует также класс под названием

`CryptoStream`, который может шифровать или дешифровать содержимое других потоков, используя любой из множества механизмов шифрования, поддерживаемых в .NET.

Один тип, много поведений

Как вы уже видели, абстрактный базовый класс `Stream` используется в самых разных сценариях. Возможно, эта абстракция даже чересчур абстрактна. Наличие таких свойств, как `CanSeek`, которые сообщают вам, можно ли использовать определенный поток определенным образом, возможно, является признаком более глубокой проблемы — кода «с душком». Этот конкретный обобщенный подход не является изобретением потоков .NET — он был популяризирован Unix и стандартной библиотекой языка программирования С уже очень давно. Проблема в том, что при написании кода, имеющего дело со `Stream`, вы можете не знать, с чем именно вы работаете.

Существует много разных способов использования `Stream`, но чаще всего можно встретить три сценария использования:

- Последовательный доступ к последовательности байтов.
- Произвольный доступ, подразумевающий эффективное кэширование.
- Доступ к некоторым базовым возможностям устройства или системы.

Как вы знаете, не все реализации `Stream` поддерживают все три модели — если `CanSeek` возвращает `false`, это исключает средний вариант. Но менее очевидно то, что, даже когда эти свойства указывают на наличие возможности, не все потоки поддерживают все эти модели использования одинаково хорошо.

Например, я работал над проектом, который использовал библиотеку для доступа к файлам в облачной службе хранения, которая умела представлять эти файлы с помощью объектов `Stream`. Это удобно, потому что вы можете передать их любому API, который работает с `Stream`. Тем не менее он был разработан для третьего стиля использования выше: каждый отдельный вызов `Read` (или `ReadAsync`) заставлял бы библиотеку отправлять HTTP-запрос службе хранения. Первоначально мы надеялись использовать это с другой библиотекой, которая знала, как анализировать файлы `Parquet` (двоичный формат хранения табличных данных, широко используемый при обработке больших объемов данных). Однако оказалось, что библиотека ожидает поток, который поддерживает второй тип доступа: она перепрыгивала туда-сюда по файлу, делая большое количество довольно коротких

операций чтения. Она отлично работала с типом `FileStream`, который я опишу позже, потому что он прекрасно поддерживает первые два режима использования. (Что касается второго стиля, то в вопросе кэширования она полагается на ОС.) Но ужасающим ударом по производительности было бы подключить `Stream` из библиотеки службы хранения непосредственно к библиотеке анализа `Parquet`.

Подобное несоответствие не всегда легко обнаружить. В этом примере свойства, указывающие на возможности, такие как `CanSeek`, не дали бы ни малейшего представления о проблеме. А приложения, использующие файлы `Parquet`, часто работают с какой-то службой удаленного хранения вместо локальной файловой системы, поэтому не было очевидной причины полагать, что эта библиотека будет исходить из того, что любой `Stream` будет использовать локальное кэширование вроде файловой системы. Технически это сработало: библиотека хранилища `Stream` усердно работала, чтобы сделать все, что от нее требуется, и код давал результат ... через какое-то время. Поэтому всякий раз, когда вы используете `Stream`, важно убедиться, что вы полностью понимаете, на какие схемы доступа он рассчитан и насколько эффективно он их поддерживает.

В некоторых случаях вам удастся преодолеть разрыв. Класс `BufferedStream` часто может принимать `Stream`, предназначенный только для третьего стиля использования, описанного выше, и адаптировать его для первого. Однако в библиотеке классов .NET нет ничего, что могло бы добавить поддержку второго стиля использования в поток, который изначально его не поддерживает. (Обычно он доступен только потокам, которые представляют что-то уже полностью размещенное в памяти или оборачивают какой-то локальный API, который выполняет кэширование, например API файловой системы ОС.) В таких случаях придется либо переосмыслить свой дизайн (например, создав локальную копию потока), либо изменить то, как используется `Stream`, либо написать специальный адаптер для кэширования. (В итоге мы написали адаптер, который расширил возможности `BufferedStream` с помощью кэширования произвольного доступа, что решило проблемы с производительностью.)

Ориентированные на текст типы

Класс `Stream` ориентирован на байты, но обычно работает с файлами, содержащими текст. Если вы хотите обрабатывать текст, хранящийся в файле

(или полученный по сети), использовать API на основе байтов неудобно, поскольку это заставляет явно учитывать все возможные варианты. Например, существует несколько соглашений о том, как представлять конец строки, — Windows обычно использует два байта со значениями 13 и 10, как и многие интернет-стандарты, такие как HTTP, но Unix-подобные системы часто используют только один байт со значением 10.

Есть также несколько популярных кодировок символов. Некоторые файлы используют один байт на символ, некоторые — два, а некоторые — переменной длины. Однобайтовых кодировок тоже немало, поэтому если вы в текстовом файле встретите байтовое значение, скажем, 163, вы не сможете узнать, что оно означает, если не обладаете информацией об используемой кодировке.

В файле с использованием однобайтовой кодировки Windows-1252 значение 163 представляет знак фунта: £. Но если файл закодирован в соответствии с ISO/IEC 8859-5 (разработан для регионов, в которых используются кириллические алфавиты), точно такой же код представляет прописную букву сербского кириллического алфавита Ђе: Ђ. И если файл использует кодировку UTF-8, этот символ будет разрешен только как часть многобайтовой последовательности, представляющей один символ.

Осведомленность об этих проблемах, конечно, является неотъемлемой частью навыков любого разработчика, но это не должно означать, что вам нужно обрабатывать каждую мелочь каждый раз, когда вы работаете с текстом. Так что .NET определяет специализированные абстракции для работы с текстом.

TextReader и TextWriter

Абстрактные классы `TextReader` и `TextWriter` представляют данные в виде последовательности значений `char`. По логике, эти классы похожи на поток, но каждый элемент в последовательности представляет собой `char` вместо `byte`. Тем не менее имеются кое-какие различия в деталях. Во-первых, существуют отдельные абстракции для чтения и записи. `Stream` содержит и то и другое, потому что это нормально, когда требуется доступ на чтение/запись к одному объекту, особенно если поток представляет файл на диске. Для байтового произвольного доступа это вполне разумно, но в случае текста подобная абстракция представляет собой проблему.

Кодировки переменной длины затрудняют поддержку произвольного доступа для записи (т. е. возможность изменять значения в любой точке последовательности). Подумайте, что получится, если взять текстовый файл UTF-8 объемом 1 ГБ, первым символом которого является \$, и заменить этот первый символ на £. В UTF-8 символ \$ занимает только один байт, а символ £ — уже два, поэтому изменение этого первого символа потребует вставки дополнительного байта в начале файла. Это будет означать сдвиг оставшегося содержимого файла — а это почти 1 ГБ данных — на один байт.

Даже произвольный доступ только для чтения является относительно дорогим. Чтобы добраться до миллионного символа в файле UTF-8, вам придется прочитать первые 999 999 символов, потому что без этого у вас нет возможности узнать, какие символы однобайтовые, а какие многобайтовые. Миллионный символ может начинаться с миллионного байта, но может и где-то в интервале от одного до четырех миллионов. Поскольку поддержка произвольного доступа с помощью текстовых кодировок переменной длины является дорогостоящей, особенно в случае записи данных, указанные текстовые типы ее не предлагают. Без произвольного доступа нет смысла объединять запись и чтение в рамках одного типа. Кроме того, разделение типов для чтения и записи устраниет необходимость проверки свойства `CanWrite` — вы знаете, что можете записывать, потому что используете `TextWriter`.

`TextReader` дает несколько способов чтения данных. Простейшей является перегрузка `Read` с нулевым аргументом, которая возвращает `int`. Она вернет `-1`, если вы достигли конца ввода, а в противном случае вернет символьное значение. (Вам нужно будет привести его к `char` после того, как вы убедились, что значение не является отрицательным.) В качестве альтернативы имеется два метода, которые похожи на метод `Read` класса `Stream` и показаны в листинге 15.4.

Листинг 15.4. Методы чтения фрагментов `TextReader`

```
public virtual int Read(char[] buffer, int index, int count) { ... }
public virtual int ReadBlock(char[] buffer, int index, int count) { ... }
```

Как и `Stream.Read`, они принимают массив, а также индекс в этом массиве и счетчик, после чего пытаются прочитать число указанных значений. Наиболее очевидное отличие от `Stream` в том, что они используют `char` вместо `byte`. Но в чем разница между `Read` и `ReadBlock`? Что ж, `ReadBlock` решает ту

же проблему, которую мне пришлось решить вручную для `Stream` в листинге 15.2: тогда как `Read` может вернуть меньше символов, чем вы запросили, `ReadBlock` не завершится, пока не станет доступно столько символов, сколько вы запросили, или пока не будет достигнут конец содержимого.

Одна из проблем обработки ввода текста связана с различными соглашениями об окончании строк, и `TextReader` способен вас от нее оградить. Его метод `ReadLine` читает всю строку ввода и возвращает ее в виде `string`. Эта строка не будет содержать символ или символы конца строки.



`TextReader` не предполагает одного конкретного соглашения о конце строки. Он принимает либо возврат каретки (символьное значение 13, которое мы записываем как `\r` в строковых литералах), либо перевод строки (10 или `\n`). И если оба символа появляются рядом, то пара целиком считается одним концом строки, несмотря на то что это два символа. Эта обработка происходит только при использовании `ReadLine` или `ReadLineAsync`. Если вы работаете непосредственно на уровне символов с помощью `Read` или `ReadBlock`, вы увидите символы конца строки во всей красе.

`TextReader` также содержит метод `ReadToEnd`, который читает входные данные полностью и возвращает их в виде единой строки. И наконец, есть `Peek`, который делает то же самое, что и метод `Read` с одним аргументом, за исключением того, что не меняет состояние читающего объекта. Он позволяет взглянуть на следующий символ, не поглощая его, поэтому в следующий раз, когда вы вызовете `Peek` или `Read`, они снова вернут тот же символ.

Что касается `TextWriter`, он содержит два перегруженных метода для записи: `Write` и `WriteLine`. Каждый из них содержит перегрузки для всех встроенных значимых типов (`bool`, `int`, `float` и т. д.). Функционально класс мог бы обойтись единственной перегрузкой, которая принимает `object`, потому что можно просто вызывать `ToString` для аргумента, но специализированные перегрузки позволяют избежать упаковки аргумента. `TextWriter` содержит метод `Flush` по той же причине, что и `Stream`.

По умолчанию `TextWriter` будет использовать последовательность конца строки по умолчанию для ОС, на которой вы работаете. В Windows это последовательность `\r\n` (13 и 10). В Linux это будет по одному `\n` в конце каждой строки. Вы можете изменить это, установив свойство `NewLine` класса записи.

Оба этих абстрактных класса реализуют `IDisposable`, потому что некоторые из конкретных производных типов записи и чтения текста являются обертками вокруг неуправляемых ресурсов или других типов удаляемых ресурсов.

Как и в случае `Stream`, эти классы содержат асинхронные версии своих методов. В отличие от `Stream`, это довольно недавнее добавление, поэтому они поддерживают только шаблон на основе задач, описанный в главе 16, который можно использовать с ключевым словом `await`, в свою очередь описанным в главе 17.

Конкретные типы для чтения и записи

Как и в случае с `Stream`, различные API в .NET могут предоставить вам объекты `TextReader` и `TextWriter`. Например, класс `Console` определяет свойства `In` и `Out`, которые обеспечивают текстовый доступ к потокам ввода и вывода процесса. Вы их еще не встречали, но неявно мы их использовали — перегрузки метода `Console.WriteLine` — это всего лишь оболочки, которые вызывают `Out.WriteLine` за вас. Аналогично методы `Read` и `ReadLine` класса `Console` — это просто перенаправление в `In.Read` и `In.ReadLine`. Имеется также `Error`, еще один `TextWriter` для записи в стандартный поток вывода ошибок. Однако есть некоторые конкретные производные от `TextReader` или `TextWriter` классы, которые вам может понадобиться создать напрямую.

StreamReader и StreamWriter

Возможно, наиболее полезными конкретными типами чтения и записи текста являются `StreamReader` и `StreamWriter`, которые оборачивают объект `Stream`. Можно передать `Stream` в качестве аргумента конструктора или просто передать строку, содержащую путь к файлу, и в этом случае они автоматически создадут `FileStream` за вас, после чего обернут его. В листинге 15.5 используется эта техника для записи текста в файл.

Листинг 15.5. Запись текста в файл с помощью `StreamWriter`

```
using (var fw = new StreamWriter(@"c:\temp\out.txt"))
{
    fw.WriteLine($"Writing to a file at {DateTime.Now}");
}
```

Имеются различные перегрузки конструктора, предлагающие более полный контроль. При передаче строки в случае использования файла со

`StreamWriter` (в отличие от `Stream`, который вы уже получили) вы можете при желании передать `bool`, указывающий, нужно ли начинать запись с нуля или же добавлять данные к существующему файлу, если он существует. (Значение `true` подразумевает добавление.) Если вы не передадите этот аргумент, добавление использовано не будет и запись начнется с самого начала. Вы также можете указать кодировку. По умолчанию `StreamWriter` будет использовать UTF-8 без метки порядка байтов (BOM), но вы можете передавать любой тип, производный от класса `Encoding`, который описан в разделе «Кодировка» на с. 774.

`StreamReader` работает так же — вы можете создать его, передав либо `Stream`, либо строку, содержащую путь к файлу, и при желании указав кодировку. Однако если вы не укажете кодировку, поведение будет немного отличаться от `StreamWriter`. В то время как `StreamWriter` по умолчанию использует UTF-8, `StreamReader` попытается выяснить кодировку содержимого потока. Он просматривает первые несколько байтов и ищет определенные особенности, которые обычно указывают на использование конкретной кодировки. Если кодированный текст начинается со спецификации Unicode, это позволяет с высокой степенью достоверности определить кодировку.

StringReader и StringWriter

Назначение классов `StringReader` и `StringWriter` аналогично `MemoryStream`: они пригодятся, когда вы работаете с API, для которого требуется либо `TextReader`, либо `TextWriter`, но работать вы хотите полностью в памяти. Если `MemoryStream` представляет API `Stream` поверх массива `byte[]`, то `StringReader` оборачивает строку, превращая ее в `TextReader`, в то время как `StringWriter` представляет собой API `TextWriter` поверх `StringBuilder`.

Один из API, которые .NET предлагает для работы с XML, а именно `XmlReader`, требует либо `Stream`, либо `TextReader`. Предположим, у вас имеется содержимое в формате XML в виде строки. Если вы передадите строку при создании нового `XmlReader`, она будет интерпретирована как URI, из которого нужно извлечь контент, но не как контент. Конструктор `StringReader`, который принимает строку, попросту оборачивает ее, превращая в содержимое, так что мы можем передать ее перегрузке `XmlReader.Create`, для которой требуется `TextReader`, как показано в листинге 15.6. (Строка, которая это делает, выделена жирным шрифтом — последующий код просто использует `XmlReader` для чтения содержимого, чтобы показать, что все работает как положено.)

Листинг 15.6. Оборачивание строки в `StringReader`

```
string xmlContent =
    "<message><text>Hello</text><recipient>world</recipient></
     message>";
var xmlReader = XmlReader.Create(new StringReader(xmlContent));
while (xmlReader.Read())
{
    if (xmlReader.NodeType == XmlNodeType.Text)
    {
        Console.WriteLine(xmlReader.Value);
    }
}
```

Что касается `StringWriter`, вы уже видели его в главе 1. Как вы помните, самый первый пример в этой книге — это юнит-тест, который проверяет, что тестируемая программа выдает ожидаемый результат (непотопляемое «Hello, world!»). Соответствующие строки повторяются в листинге 15.7.

Листинг 15.7. Захват вывода консоли в `StringWriter`

```
var w = new System.IO.StringWriter();
Console.SetOut(w);
```

Так же как в листинге 15.6 используется API, который ожидает `TextReader`, в листинге 15.7 используется тот, который требует `TextWriter`. Мне требуется записать в память все, что было передано процедуре записи (т. е. все вызовы `Console.Write` и `Console.WriteLine`), чтобы мой тест мог все это проверить. Вызов `SetOut` дает возможность предоставить `StringWriter`, который используется для вывода в консоль.

Кодировка

Как я упоминал ранее, если вы используете `StreamReader` или `StreamWriter`, то им необходимо знать, какую кодировку символов использует базовый поток. Тогда они смогут правильно преобразовывать байты в потоке, а также типы символов или строк .NET. Для этого в пространстве имен `System.Text` определен абстрактный класс `Encoding` с различными публичными конкретными производными типами, основанными на кодировке. В их число входят `ASCIIEncoding`, `UTF7Encoding`, `UTF8Encoding`, `UTF32Encoding` и `UnicodeEncoding`.

Большинство этих имен типов не требуют пояснений, потому что названы в соответствии со стандартными кодировками, которые они и представляют — ASCII или UTF-8. Единственный класс, который требует пояснения, —

это `UnicodeEncoding`. В конце концов, UTF-7, UTF-8 и UTF-32 являются кодировками Unicode, так для чего же нужен этот класс? Когда в первой версии Windows NT представили поддержку Unicode, было принято одно неудачное соглашение: в документации и различных именах API термин «Unicode» использовался для обозначения 2-байтовой кодировки с прямым порядком байтов, которая является лишь одной из многих возможных схем кодирования, все из которых вполне можно обозначить как «Unicode» той или иной формы².

Класс `UnicodeEncoding` назван так, чтобы соответствовать этому историческому соглашению, хотя даже в этом случае название все еще немного сбивает с толку. Кодировка, которую Win32 API называет «Unicode», по сути является UTF-16LE, но класс `UnicodeEncoding` также способен поддерживать UTF-16BE с обратным порядком байтов.

Базовый класс `Encoding` определяет статические свойства, которые возвращают экземпляры всех упомянутых выше типов кодировки, поэтому если вам нужен объект, представляющий определенную кодировку, то вместо конструирования нового объекта вы просто пишете `Encoding.ASCII` или `Encoding.UTF8` и т. д. Существует два свойства типа `UnicodeEncoding`: свойство `Unicode` возвращает объект, настроенный для работы с UTF-16LE, а `BigEndianUnicode` — для UTF-16BE.

Для различных кодировок `Unicode` эти свойства будут возвращать объекты кодировки, которые сообщат `StreamWriter` о необходимости добавить спецификацию в начале вывода. Основная цель спецификации состоит в том, чтобы дать программному обеспечению возможность автоматически определить, содержит ли кодировка прямую или обратную последовательность байтов. (Вы также можете использовать ее для распознавания UTF-8, потому в этом случае спецификация будет отличаться от других.) Если вы знаете, что будете использовать кодировку, уточняющую порядок байтов (например, UTF-16LE), то в спецификации нет необходимости, потому что порядок известен, но спецификация `Unicode` определяет гибкие форматы, в которых закодированные байты способны объявлять используемый порядок.

² Если вдруг вы не сталкивались с этим термином, в представлениях с прямым порядком (little-endian) байтов многобайтовые значения начинаются с байтов младшего разряда, поэтому значение 0x1234 в 16-битном порядке с прямым порядком байтов будет выглядеть как 0x34, 0x12, тогда как версия значения с обратным порядком байтов (big-endian) будет выглядеть как 0x12, 0x34. Прямой порядок выглядит перевернутым, но это родной формат процессоров Intel.

док с помощью расположенной в начале спецификации, символа с кодовой точкой Unicode U+FEFF. 16-разрядная версия этой кодировки называется просто UTF-16, и вы можете определить порядок следования в каком-либо конкретном наборе байтов в кодировке UTF-16, посмотрев, начинается он с 0xFE, 0xFF или же с 0xFF, 0xFE.

Как упоминалось ранее, если не указать кодировку при создании `StreamWriter`, по умолчанию будет использована UTF-8 без спецификации, что отличается от `Encoding.UTF8`, которая создает спецификацию. `StreamReader` еще интереснее: если вы не укажете кодировку, он попытается ее определить. Таким образом, .NET может обрабатывать автоматическое обнаружение порядка следования байтов, как того требует спецификация Unicode для UTF-16 и UTF-32, просто для этого не нужно указывать какую-либо конкретную кодировку при создании `StreamReader`. Он будет искать спецификацию и, если найдет, будет использовать соответствующую кодировку Unicode; в противном случае предполагается кодировка UTF-8.



Хотя Unicode определяет схемы кодировок, которые позволяют обнаруживать порядок байтов, невозможно создать объект `Encoding`, который работает таким образом, — он всегда должен иметь определенный порядок байтов. Хотя `Encoding` и определяет, должна ли быть записана спецификация при записи данных, это не влияет на поведение при чтении данных — в данном случае всегда предполагается последовательность, указанная при создании `Encoding`. Это означает, что свойство `Encoding.UTF32`, возможно, названо неправильно — оно всегда интерпретирует данные как данные с прямым порядком байтов, хотя спецификация Unicode допускает использование UTF-32 как с прямым, так и с обратным порядком. `Encoding.UTF32` — это на самом деле UTF-32LE.

UTF-8 — это популярная кодировка. Если ваш основной язык — английский, это особенно удобное представление, потому что, если вы используете только символы, доступные в ASCII, каждый символ будет занимать один байт, а текст будет иметь те же байтовые значения, что и в случае кодировки ASCII. Но в отличие от ASCII, вы не ограничены 7-битным набором символов. Вам доступны все кодовые точки Unicode — нужно просто использовать многобайтовые представления для всего, что находится за пределами диапазона ASCII. Однако, несмотря на свое распространение, UTF-8 — это не единственная популярная 8-битная кодировка.

Кодировки на основе кодовой страницы

Windows, как и DOS до нее, долгое время поддерживала 8-битные кодировки, расширяющие ASCII. ASCII — это 7-битная кодировка, а это означает, что с 8-битными байтами у вас есть 128 «запасных» значений для других символов. Этого и близко недостаточно для охвата каждого символа каждой локали, но внутри конкретной страны этого часто бывает достаточно (хотя и не всегда — во многих дальневосточных странах требуется более 8 бит на символ). Но в каждой стране имеется разный набор отличных от ASCII символов в зависимости от того, какие символы с акцентами популярны в данном регионе и нужно ли использование нелатинского алфавита. Таким образом, существуют разные *кодовые страницы* для разных региональных настроек. Например, кодовая страница 1253 использует значения в диапазоне 193–254 для определения символов из греческого алфавита (заполняя оставшиеся значения, не входящие в ASCII, полезными символами, такими как символы различных валют). Кодовая страница 1255 вместо этого определяет символы иврита, в то время как 1256 — арабские символы в верхнем диапазоне (и у этих конкретных кодовых страниц есть некоторые общие черты, такие как использование 128 для символа евро, €, и 163 для знака фунта, £).

Одна из наиболее часто встречающихся кодовых страниц — 1252, потому что она по умолчанию используется для англоязычных региональных настроек в Windows. В ней нет символов, отличных от латинского алфавита; вместо этого она использует верхний диапазон для полезных символов и различных акцентированных версий латинского алфавита, которые дают возможность представить широкий спектр западноевропейских языков.

Вы можете создать кодировку для кодовой страницы, вызвав метод `Encoding.GetEncoding` и передав номер кодовой страницы. (Конкретный тип возвращаемого вами объекта часто не относится к тем, которые я перечислял ранее. Этот метод может возвращать `не-public` типы, которые являются производными от `Encoding`.) Листинг 15.8 использует эту технику для записи текста, содержащего знак фунта, в файл с использованием кодовой страницы 1252.

Листинг 15.8. Запись с помощью кодовой страницы Windows 1252

```
using (var sw = new StreamWriter("Text.txt", false,
                                Encoding.GetEncoding(1252)))
{
    sw.WriteLine("£100");
}
```

Символ £ будет представлен как один байт со значением 163. В кодировке по умолчанию, UTF-8, он был бы представлен двумя байтами со значениями 194 и 163 соответственно.

Непосредственное использование кодировок

`TextReader` и `TextWriter` — не единственный способ использования кодировки. Объекты, представляющие кодировки (такие, как `Encoding.UTF8`), определяют различные члены. Например, метод `GetBytes` преобразует строку непосредственно в массив `byte[]`, а метод `GetString` выполняет обратное преобразование.

Также можно узнать, сколько данных произведут эти преобразования. `GetByteCount` сообщит вам, какой размер массива `GetBytes` произведет из данной строки, а `GetCharCount` — сколько символов получится из конкретного массива. С помощью `GetMaxByteCount` вы также можете вычислить верхний предел требуемого места, не зная точного текста. Вместо `string` метод принимает число, которое интерпретируется как длина строки; поскольку строки .NET используют UTF-16, это означает, что данный API отвечает на вопрос: «Если у меня столько-то кодовых единиц UTF-16, какое наибольшее количество кодовых единиц потребуется для представления этого же текста в целевой кодировке?» Это может привести к значительному завышению оценки в случае использования кодировки переменной длины. Например, в случае UTF-8 `GetMaxByteCount` умножает длину входной строки на три и добавляет дополнительные 3 байта для обработки краевого случая, который может возникнуть с суррогатными символами³. Он дает правильное описание наихудшего возможного случая, но текст, содержащий любые символы, для которых не требуется 3 байта в UTF-8 (т. е. любой текст на английском или любых других языках, использующих латинский алфавит, а также любой текст на греческом языке, кириллице, иврите или арабском языке), потребует значительно меньше места, чем предсказывает `GetMaxByteCount`.

Некоторые кодировки могут содержать *пreamble*, отличительную последовательность байтов, которая, будучи найдена в начале некоторого текста, указывает на то, что вы, вероятно, наткнетесь именно на эту кодировку. Это может пригодиться, если вы пытаетесь определить используемую коди-

³ Некоторые символы Unicode могут занимать до 4 байтов в UTF-8, поэтому умножение на три может оказаться занижением. Однако все такие символы требуют двух байтов в UTF-16. Никакой символ в .NET не требует более 3 байтов в UTF-8.

ровку, не зная ее заранее. Различные кодировки Unicode возвращают свою спецификацию в виде преамбулы, которую вы всегда можете получить с помощью метода `GetPreamble`.

Класс `Encoding` определяет свойства экземпляра, содержащие информацию о кодировке. `EncodingName` возвращает удобочитаемое имя кодировки, но вместе с тем доступно еще два. Свойство `WebName` возвращает стандартное имя для кодировки, зарегистрированное в реестре Internet Assigned Numbers Authority (IANA), который регулирует стандартные имена и номера для сущностей в интернете, таких как MIME-типы. Некоторые протоколы, такие как HTTP, иногда помещают имена кодировки в заголовки, и это текст, который вы должны использовать в подобной ситуации. Два других имени, `BodyName` и `HeaderName`, несколько более таинственны и используются только для электронной почты — есть разные соглашения о том, как определенные кодировки представлены в теле и заголовках электронной почты.

Файлы и каталоги

Абстракции, которые я пока показал в этой главе, носят довольно общий характер — вы можете писать код, который использует `Stream`, не имея необходимости знать, откуда берутся или куда отправляются байты, это же касается `TextReader` и `TextWriter` — они не требуют какого-либо конкретного источника или места назначения для своих данных. Это полезно и позволяет писать код, который применим в различных сценариях. Например, основанный на потоке `GzipStream` может сжимать или распаковывать данные из файла, из сетевого соединения или из любого другого потока. Однако бывают случаи, когда вы знаете, что будете иметь дело с файлами, и хотите получить доступ к функциям, специфичным для файлов. В этом разделе описаны классы для работы с файлами и файловой системой.

Класс `FileStream`

Класс `FileStream` является производным от `Stream` и представляет файл из файловой системы. Я уже мимоходом использовал его несколько раз. Он добавляет относительно немного членов к тем, которые определены базовым классом. Методы `Lock` и `Unlock` дают возможность получения исключительного доступа к определенным диапазонам байтов при использовании одного файла из нескольких процессов. Свойство `Name` сообщает вам имя файла.

Конструкторы `FileStream` позволяют много чего контролировать — помимо помеченных атрибутом `[Obsolete]` имеется не менее девяти перегрузок конструктора⁴. Способы создания `FileStream` делятся на две группы: те, где у вас уже есть дескриптор файла ОС, и те, где еще нет. Если у вас уже откуда-то есть дескриптор, вы должны сообщить `FileStream`, дает ли этот дескриптор доступ на чтение, запись или на чтение/запись. Это можно сделать, передав значение из перечисления `FileAccess`. Другие перегрузки позволяют при необходимости указать размер буфера, который вы хотели бы использовать при чтении или записи, а также флаг, указывающий, был ли открыт дескриптор для перекрывающегося ввода-вывода, механизма Win32 для поддержки асинхронной операции. (Конструкторы, которые не принимают этот флаг, предполагают, что вы не запрашивали перекрывающийся ввод-вывод при создании дескриптора файла.)

Чаще всего используются другие конструкторы, в которых `FileStream` использует API ОС для создания дескриптора файла от вашего имени. У вас есть возможность точного управления тем, как вы хотите, чтобы это было сделано. Как минимум вы должны указать путь к файлу и значение из перечисления `FileMode`. В табл. 15.1 приведены значения, содержащиеся в этом перечислении, а также описаны действия конструктора `FileStream` для каждого значения в ситуациях, когда означенный файл уже существует или не существует.

Таблица 15.1. Перечисление `FileMode`

Значение	Поведение, если файл существует	Поведение, если файл не существует
<code>CreateNew</code>	Вызывает <code> IOException</code>	Создает новый файл
<code>Create</code>	Заменяет существующий файл	Создает новый файл
<code>Open</code>	Открывает существующий файл	Вызывает <code> FileNotFoundException</code>
<code>OpenOrCreate</code>	Открывает существующий файл	Создает новый файл
<code>Truncate</code>	Заменяет существующий файл	Вызывает <code> FileNotFoundException</code>
<code>Append</code>	Открывает существующий файл, устанавливая <code>Position</code> в конец файла	Создает новый файл

При желании также можно указать `FileAccess`. Если вы этого не сделаете, `FileStream` использует `FileAccess.ReadWrite`, если, конечно, вы не выбра-

⁴ Четыре перегрузки устарели, когда в .NET 2.0 появился новый способ представления дескрипторов ОС. На тот момент устарели перегрузки, которые принимают `IntPtr`, и их заменили новые, принимающие `SafeFileHandle`.

ли элемент `Append` из `FileMode`. В файлы, открытые в режиме добавления, можно только записывать, поэтому в этом случае `FileStream` выберет `Write` (Если при открытии в режиме `Append` вы явно передадите `FileAccess`, запрашивающий что-либо кроме `Write`, конструктор выдаст исключение `ArgumentException`.)

Кстати, когда я описываю каждый следующий аргумент конструктора в этом разделе, следует помнить, что соответствующая перегрузка также будет принимать и все ранее описанные (за исключением аргумента `useAsync`, который появляется только в одном конструкторе). Как показано в листинге 15.9, большинство этих конструкторов выглядят так же, как предыдущие, но с одним дополнительным аргументом.

Листинг 15.9. Конструкторы `FileStream`, принимающие путь

```
public FileStream(string path, FileMode mode)
public FileStream(string path, FileMode mode, FileAccess access)
public FileStream(string path, FileMode mode, FileAccess access,
                  FileShare share)
public FileStream(string path, FileMode mode, FileAccess access,
                  FileShare share, int bufferSize);
public FileStream(string path, FileMode mode, FileAccess access,
                  FileShare share, int bufferSize, bool useAsync);
public FileStream(string path, FileMode mode, FileAccess access,
                  FileShare share, int bufferSize, FileOptions options);
```

Если вы передаете аргумент типа `FileShare`, вы можете указать, нужен ли вам исключительный доступ к файлу или вы готовы позволить другим процессам (или другому коду в вашем процессе) открывать файл совместно. По умолчанию вы получаете общий доступ для чтения, что означает, что разрешено несколько одновременных операций чтения, но если что-либо открывает файл с доступом для записи или чтения/записи, другие дескрипторы не могут быть открыты в то же время. Что еще более странно, вы можете включить совместное использование записи, при котором может быть одновременно активно любое количество дескрипторов с доступом для записи, но никто не сможет читать из файла, пока не будут освобождены все остальные дескрипторы. Существует значение `ReadWrite`, которое позволяет одновременные чтение и запись. Вы также можете передать `Delete`, указывая тем самым, что не возражаете, если кто-то еще попытается удалить файл, пока он вами открыт. Конечно, нужно быть готовым к тому, что вы получите исключения ввода-вывода, если попытаетесь использовать файл после удаления, но иногда усилия могут

быть оправданы; в противном случае попытки удалить открытый файл будут заблокированы.

Для открытия нескольких дескрипторов все стороны должны договориться о совместном использовании. Если программа А использует для открытия файла `FileShare.ReadWrite`, после чего программа Б при попытке открыть файл для чтения и записи передаст `FileShare.None`, программа Б получит исключение, поскольку хотя А была готова к совместному использованию, Б не была, поэтому требования Б не могут быть выполнены. Если бы программе Б удалось открыть файл первой, у нее бы это получилось, а запрос А оказался бы безуспешным.



В Unix меньше комплексных механизмов блокировки файлов, нежели в Windows, поэтому в этих средах подобная семантика блокировки часто будет сопоставляться с чем-то более простым. Кроме того, блокировки файлов лишь рекомендуются в Unix, т. е. процессы могут просто игнорировать их, если им это нужно.

Следующий фрагмент информации, который мы можем передать, — это размер буфера. Он определяет размер блока, который `FileStream` будет использовать при чтении и записи на диск. По умолчанию это 4096 байт. В большинстве сценариев это значение работает просто отлично, но, если вы обрабатываете очень большие объемы данных с диска, больший размер буфера может обеспечить лучшую пропускную способность. Однако, как и в других случаях, касающихся производительности, вы должны измерить эффект подобного изменения, чтобы определить, стоит ли оно того, — в ряде случаев вы не увидите никакой разницы в скорости обработки данных и попросту будете использовать немного больше памяти, чем необходимо.

Флаг `useAsync` позволяет определить, открыт ли дескриптор файла способом, оптимизированным для объемных асинхронных операций чтения и записи. (В Windows он открывает файл для перекрывающегося ввода-вывода функции Win32, поддерживающей асинхронные операции.) Если вы читаете данные относительно большими кусками и используете асинхронные API потока, то с этим флагом вы, как правило, получаете более высокую производительность. Однако этот режим фактически увеличивает издержки при чтении данных по несколько байтов за раз. Если код, обращающийся к файлу, особенно чувствителен к производительности, стоит попробовать

оба параметра, чтобы определить, какой из них лучше подходит для вашей рабочей нагрузки.

Следующий аргумент, который вы можете передать, имеет тип `FileOptions`. Если вы внимательно посмотрите на листинг 15.9, вы заметите, что каждая из перегрузок добавляет еще один аргумент, но данный аргумент типа `FileOptions` заменяет аргумент `useAsync` типа `bool`. Это происходит потому, что одним из параметров, которые вы можете указать с помощью `FileOptions`, является асинхронный доступ. `FileOptions` — это перечисление флагов, поэтому вы можете указать комбинацию любых содержащихся в нем флагов, которые описаны в табл. 15.2.

Будьте осторожны с флагом `WriteThrough`. Хотя он и работает так, как заявлено, он может не дать желаемого эффекта, поскольку некоторые жесткие диски откладывают запись с целью повышения производительности. (Многие жесткие диски имеют собственную оперативную память, что позволяет им очень быстро получать данные с компьютера и сообщать об операциях записи как выполненных до фактического сохранения данных.) Флаг `WriteThrough` гарантирует, что при удалении или сбросе потока все записанные вами данные будут отправлены на накопитель, но накопитель не обязательно тут же запишет эти данные, поэтому вы все равно можете их потерять в случае сбоя питания. Точное поведение будет зависеть от того, каким образом вы попросили ОС настроить диск.

Таблица 15.2. Флаги `FileOptions`

Флаг	Значение
<code>WriteThrough</code>	Отключает буферизацию записи в ОС, поэтому при сбросе потока данные передаются прямо на диск
<code>Asynchronous</code>	Задает использование асинхронного ввода-вывода
<code>RandomAccess</code>	Подсказывает кэшу файловой системы, что вы будете осуществлять поиск, а не чтение или запись данных по порядку
<code>SequentialScan</code>	Подсказывает кэшу файловой системы, что вы будете читать или записывать данные по порядку
<code>DeleteOnClose</code>	Сообщает <code>FileStream</code> , что при вызове <code>Dispose</code> нужно удалить файл
<code>Encrypted</code>	Шифрует файл, чтобы его содержимое не могло быть прочитано другими пользователями

Хотя `FileStream` дает контроль над содержимым файла, некоторые требуемые операции, возможно, окажутся слишком громоздкими или вообще

не будут поддерживаться `FileStream`. Например, вы можете скопировать файл с использованием этого класса, но это не так просто, как могло бы быть, а кроме того, `FileStream` не предлагает никакого способа удаления файла. Поэтому библиотека классов .NET содержит отдельный класс для подобных операций.

Класс `File`

Статический класс `File` предоставляет методы для выполнения различных операций с файлами. Метод `Delete` удаляет указанный файл из файловой системы. С помощью метода `Move` можно переместить или просто переименовать файл. Существуют методы для получения информации о атрибутов, которые файловая система хранит для каждого файла, например `GetCreationTime`, `GetLastAccessTime`, `GetLastWriteTime` и `GetAttributes`⁵. (Последний из них возвращает значение `FileAttributes`, которое является типом перечисления флагов, указывающим, является ли файл доступным только для чтения, скрытым, системным и т. д.)

Метод `Encrypt` в некоторой степени пересекается с `FileStream` — как вы видели ранее, вы можете запросить, чтобы файл сохранялся с шифрованием при создании. Однако `Encrypt` может работать с файлом, который уже был создан без шифрования, по сути тут же его зашифровывая. (Поддерживается только в Windows и только на дисках, где подобное поддерживает файловая система. Процедура будет выдавать `PlatformNotSupportedException` в других операционных системах и `NotSupportedException` в Windows, если шифрование для указанного файла недоступно. Эффект похож на включение шифрования через окно свойств файла в проводнике Windows.) Вы также можете превратить зашифрованный файл в незашифрованный, вызвав `Decrypt`.

Все остальные методы, предоставляемые `File`, просто предлагают несколько более удобные способы выполнения действий, которые можно проделать вручную с помощью `FileStream`. Метод `Copy` создает копию файла, и, хотя вы можете сделать это с помощью метода `CopyTo` в `FileStream`, `Copy` устраняет ряд неудобств. Например, он гарантирует, что в целевой файл будут перенесены такие атрибуты, как доступность только для чтения и шифрование.

⁵ Все они возвращают `DateTime` относительно текущего часового пояса компьютера. Каждый из этих методов имеет эквивалент, который возвращает время относительно нулевого часового пояса (например, `GetCreationTimeUtc`).



Не стоит вызывать `Decrypt` перед чтением зашифрованного файла. При входе в систему под той же учетной записью пользователя, под которой осуществлялось шифрование, вы можете прочитать его содержимое обычным способом — зашифрованные файлы выглядят так же, как обычные, потому что Windows автоматически расшифровывает содержимое при чтении. Цель этого механизма шифрования состоит в том, что, если какой-то другой пользователь сможет получить доступ к файлу (например, если он находится на украденном диске), содержимое будет казаться случайным набором данных. `Decrypt` удаляет это шифрование, так что любой, кто сможет получить доступ к файлу, сможет просмотреть и его содержимое.

Метод `Exists` позволяет перед попыткой открыть файл узнать, существует ли он. По большому счету, вам это не нужно, потому что `FileStream` вызовет исключение `FileNotFoundException`, когда вы попытаетесь открыть несуществующий файл, но `Exists` позволяет вам избежать исключений. Это может пригодиться, если нужно обращаться к файлу очень часто — исключения в таком случае обойдутся дорого. Тем не менее следует быть осторожным с этим методом; одно то, что `Exists` вернет `true`, не гарантирует, что вы не получите исключение `FileNotFoundException`. Всегда возможно, что между проверкой и попыткой открыть файл другой процесс может его удалить. Кроме того, файл может находиться в общей сетевой папке, а вы можете потерять подключение к сети. Соответственно вы всегда должны быть готовы к исключениям при доступе к файлам, даже если вы пытались их избежать.

`File` предлагает множество вспомогательных методов для упрощения открытия или создания файлов. Метод `Create` просто создает за вас `FileStream`, передавая подходящие значения `FileMode`, `FileAccess` и `FileShare`. В листинге 15.10 показано, как его использовать, а также как будет выглядеть эквивалентный код без использования `Create`. Метод `Create` предоставляет перегрузки, позволяющие вам указать размер буфера, `FileMode` и `FileSecurity`, а также подставляет за вас остальные аргументы.

Методы `OpenRead` и `OpenWrite` класса `File` похожим образом позволяют избавиться от всего лишнего, когда вы хотите открыть существующий файл для чтения или открыть или создать файл для записи. Существует также метод `Open`, который требует передачи `FileMode`. Он не слишком полезен, так как очень похож на перегрузку конструктора `FileStream`, которая также принимает только путь и режим, автоматически предоставляя другие параметры. Довольно необоснованное отличие состоит в том, что конструктор

`FileStream` по умолчанию использует значение `FileShare.Read`, а метод `File.Open` — `FileShare.None`.

Листинг 15.10. `File.Create` против `new FileStream`

```
using (FileStream fs = File.Create("foo.bar"))
{
    ...
}

// Эквивалентный код без использования класса File
using (var fs = new FileStream("foo.bar", FileMode.Create,
    FileAccess.ReadWrite, FileShare.None))
{
    ...
}
```

`File` также содержит несколько ориентированных на текст вспомогательных процедур. Самый простой метод, `OpenText`, открывает файл для чтения текста и не имеет особой ценности, поскольку делает то же самое, что и конструктор `StreamReader`, который принимает один строковый аргумент. Единственная причина использовать его — если вам нравится то, как он преобразовывает код. Если ваш код интенсивно использует вспомогательные процедуры `File`, вы можете использовать этот метод для идиоматической согласованности.

Некоторые из методов, предоставляемых `File`, ориентированы на текст. Они позволят улучшить код, показанный в листинге 15.11. Там добавляется строка в файл журнала.

Листинг 15.11. Добавление к файлу с помощью `StreamWriter`

```
static void Log(string message)
{
    using (var sw = new StreamWriter(@"c:\temp\log.txt", true))
    {
        sw.WriteLine(message);
    }
}
```

Одна из проблем заключается в том, что не так легко с ходу понять, как открывается `StreamWriter`, — что означает аргумент `true`? А он говорит `StreamWriter`, что мы хотим, чтобы базовый `FileStream` был открыт в режиме добавления. Листинг 15.12 делает то же самое — он использует `File.AppendText`, который приводит к вызову того же конструктора `FileStream`. И хотя ранее я пренебрежительно отзывался о `File.OpenText` из-за его бесполезности, я считаю,

что `File.AppendText`, в отличие от `File.OpenText`, привел к значительному улучшению читаемости кода. По сравнению с листингом 15.11, гораздо проще понять, что листинг 15.12 будет добавлять текст в файл. Но, поскольку в C# была добавлена поддержка именованных аргументов, `AppendText` теперь выглядит менее полезным — мы могли бы просто поименовать аргумент `append` в листинге 15.11 для аналогичного улучшения читаемости.

Если нужно добавить текст в файл и тут же закрыть его, есть еще более простой способ. Как показано в листинге 15.13, мы можем еще больше все упростить с помощью вспомогательного метода `AppendAllText`.

Но здесь следует проявить осторожность. Это не совсем то же самое, что делает листинг 15.12. Тот пример для добавления текста использовал `WriteLine`, но листинг 15.13 эквивалентен использованию только `Write`. Таким образом, если бы вы вызвали метод `Log` в листинге 15.13 несколько раз, вы бы получили одну длинную строку в вашем файле, если только записанные строки не содержали символов конца строки. Если вам нужно работать со строками, то для этого есть метод `AppendAllLines`, который принимает коллекцию строк и добавляет каждую в качестве новой строки в конец файла. Листинг 15.14 использует этот метод для добавления полной строки в каждом вызове.

Листинг 15.12. Создание добавляющего `StreamWriter` с помощью `File.AppendText`

```
static void Log(string message)
{
    using (StreamWriter sw = File.AppendText(@"c:\temp\log.txt"))
    {
        sw.WriteLine(message);
    }
}
```

Листинг 15.13. Добавление в файл одной строки

```
static void Log(string message)
{
    File.AppendAllText(@"c:\temp\log.txt", message);
}
```

Листинг 15.14. Добавление одной строки в файл

```
static void Log(string message)
{
    File.AppendAllLines(@"c:\temp\log.txt", new[] { message });
}
```

Так как `AppendAllLines` принимает `IEnumerable<string>`, вы можете использовать его для добавления любого количества строк. Но метод без проблем добавит и одну строку, если это то, что вам требуется. `File` также определяет методы `WriteAllText` и `WriteAllLines`, которые работают очень похоже, но если файл по указанному пути уже существует, то они его заменяют.

Он также содержит некоторые связанные и ориентированные на текст методы для чтения содержимого файлов. `ReadAllText` эквивалентен созданию `StreamReader` и вызову его метода `ReadToEnd`, возвращая все содержимое файла в виде одной строки. `ReadAllBytes` извлекает весь файл в массив `byte[]`. `ReadAllLines` считывает весь файл как массив `string[]`, помещая каждую строку из файла в отдельный элемент массива. На первый взгляд `ReadLines` очень на него похож. Он обеспечивает доступ ко всему файлу в виде `IEnumerable<string>` с одним элементом на каждую строку, но разница в том, что он работает медленно — в отличие от других методов, которые я описал в этом параграфе, он заранее не читает файл целиком в память, поэтому в случае очень больших файлов наилучшим выбором будет `ReadLines`. Он не только потребляет меньше памяти, но и позволяет вашему коду начать работу быстрее — вы можете начать обрабатывать данные, как только первая строка будет прочитана с диска, тогда как ни один из остальных методов не завершится, пока не прочитает файл полностью.

Класс `Directory`

Так же как `File` — это статический класс, предлагающий методы для выполнения операций с файлами, `Directory` — это статический класс, содержащий методы для работы с каталогами. Некоторые из методов очень похожи на те, которые определяет `File`; например, в нем имеются методы получения и установки времени создания, получения времени последнего доступа и времени последней записи, а также методы `Move`, `Exists` и `Delete`. В отличие от `File`, `Directory.Delete` имеет две перегрузки. Одна принимает только путь и работает, только если каталог пуст. Другая принимает `bool`, который при значении `true` удалит все в папке, рекурсивно удаляя вложенные папки и содержащиеся в них файлы. Используйте ее с осторожностью.

Конечно, есть и методы, направленные только на работу с каталогами. `GetFiles` принимает путь к каталогу и возвращает массив `string[]`, содержащий полный путь каждого файла в этом каталоге. У метода есть перегрузка, которая позволяет вам указать шаблон, согласно которому

фильтруются результаты, и еще одна, которая принимает шаблон, а также флаг, который позволяет запрашивать рекурсивный поиск всех подпапок. Листинг 15.15 использует ее, чтобы найти все файлы с расширением .jpg в моей папке Pictures. (Очевидно, что если вас не зовут Иэн, следует изменить этот путь, чтобы он соответствовал имени вашей учетной записи и работал на компьютере.) Опять же, в реальном приложении вы должны получить этот путь, используя технику, показанную в разделе «Известные папки» на с. 793.

Существует аналогичный метод `GetDirectories`, предлагающий те же три перегрузки, но вместо файлов возвращающий подкаталоги в указанном каталоге. И есть метод `GetFileSystemEntries`, с теми же тремя перегрузками, который возвращает как файлы, так и папки.

Листинг 15.15. Рекурсивный поиск файлов определенного типа

```
foreach (string file in Directory.GetFiles(@"c:\users\ian\Pictures",
                                         "*.jpg",
                                         SearchOption.AllDirectories))
{
    Console.WriteLine(file);
}
```

Существуют также методы `EnumerateFiles`, `EnumerateDirectories` и `EnumerateFileSystemEntries`, которые делают то же самое, что и три метода `GetXxx`, но возвращают `IEnumerable<string>`. Это перечисление с отложенным вычислением, поэтому вы можете начать обработку результатов немедленно, не дожидаясь всех результатов в виде одного большого массива.

Класс `Directory` определяет методы, относящиеся к текущему каталогу процесса (т. е. тому, который используется каждый раз, когда вы вызываете API доступа к файлам без указания полного пути). `GetCurrentDirectory` возвращает путь, а `SetCurrentDirectory` его задает.

Вы также можете создавать новые каталоги. Метод `CreateDirectory` принимает путь и пытается создать столько каталогов, сколько необходимо для существования пути. Таким образом, если вы передадите `C:\new\dir\here`, а каталог `C:\new` отсутствует, он создаст три новых каталога: сначала будет создан `C:\new`, затем `C:\new\dir`, а затем `C:\new\dir\here`. Если запрашиваемая папка уже существует, это не будет расценено как ошибка, а просто приведет к завершению работы метода.

`GetDirectoryRoot` обрезает путь к каталогу до имени диска или другого корня, такого как имя общего сетевого ресурса. Например, в Windows, если вы передадите `C:\temp\logs`, метод вернет `C:\`, а если передадите `\someserver\myshare\dir\test`, то он вернет `\someserver\myshare`. Этот вид обрезки строк, при котором путь разбивается на составные части, оказался достаточно востребованным, чтобы появился класс, предназначенный для такого рода операций.

Класс Path

Статический класс `Path` содержит полезные средства для работы со строками, содержащими имена файлов. Некоторые извлекают фрагменты пути к файлу, например имя содержащей папки или расширение. Некоторые объединяют строки для создания новых путей. Большинство из этих методов просто выполняют обработку строк и не требуют наличия самих файлов или каталогов, на которые указывают пути. Тем не менее есть несколько методов, которые выходят за рамки работы со строками. Например, `Path.GetFullPath` будет использовать текущий каталог, если ему не передать в качестве аргумента абсолютный путь. Но так будут делать только методы, которые вынуждены использовать реальные местоположения.

Метод `Path.Combine` решает вопросы, связанные с объединением папок и имен файлов. Если у вас есть имя папки, `C:\temp`, и имя файла, `log.txt`, то, передав оба пути в `Path.Combine`, вы получите `C:\temp\log.txt`. Также это сработает, если передать `C:\temp\` в качестве первого аргумента, поэтому одна из проблем, которую решает метод, заключается в выяснении того, нужно ли ему передавать дополнительный символ `\`. Если второй путь является абсолютным, метод это обнаруживает и просто игнорирует первый путь. Поэтому если вы передадите `C:\temp` и `C:\logs\log.txt`, результатом будет `C:\logs\log.txt`. Хотя это может показаться тривиальным, на удивление легко получить неправильную комбинацию пути к файлу, если пытаться составить ее самостоятельно объединением строк, поэтому всегда стоит избегать этого искушения и использовать `Path.Combine`.

При передаче пути к файлу метод `GetDirectoryName` удаляет часть с именем файла и просто возвращает каталог. Этот метод служит хорошей иллюстрацией того, почему следует помнить, что большинство членов класса `Path` не обращаются к файловой системе. Если вы не учли это, то будете ожидать, что при передаче `GetDirectoryName` только имени папки (например, `C:\Program Files`), он поймет, что это папка, и вернет ту же строку, но на самом деле он

вернет только C:\. Имя `Program Files` является подходящим именем как для файла, так и для каталога, и поскольку метод `GetDirectoryName` не проверяет диск и ожидает, что ему будет передан путь, содержащий имя файла, в данном случае он решит, что это файл. Этот метод по сути ищет последний символ / или \ и возвращает всю строку до него. (Таким образом, если передать имя папки с конечным символом \, например C:\Program Files\, метод вернет C:\Program Files. Опять же, весь смысл этого API заключается в удалении имени файла из полного пути к файлу. Если у вас уже есть строка лишь с именем папки, не следует вызывать этот API.)



Поведение .NET Core зависит от платформы, когда дело касается путей. В системах семейства Unix в качестве разделителя каталогов используется только символ /, поэтому различные методы в `Path`, ожидающие, что пути содержат каталоги, будут обрабатываться только / в качестве разделителей. Windows использует в качестве разделителя символ \, обычно допуская символ / в качестве замены, и `Path` следует этому правилу. Таким образом, `Path.Combine ("/x/y", "/z.txt")` даст одинаковые результаты в Windows и Linux, а вот `Path.Combine ("@"\x\y", @"\z.txt")` — нет. Если путь начинается с буквы диска, то в Windows — это абсолютный путь. Unix же не распознает буквы диска. В Linux или macOS примеры из предыдущего абзаца приведут к странным результатам, потому что в этих системах все эти пути будут рассматриваться как относительные. Если удалить буквы дисков и заменить \ на /, результаты станут такими, как вы ожидаете.

Метод `GetFileName` возвращает только имя файла (включая расширение, если оно есть). Как и `GetDirectoryName`, метод ищет последний символ разделителя каталогов, но возвращает текст, который расположен после него, а не перед ним. Повторюсь, он не обращается к файловой системе, а работает исключительно со строками (хотя, как и во всех этих операциях, он берет в расчет правила локальной системы для определения того, что считать разделителем каталогов или абсолютным путем). `GetFileNameWithoutExtension` работает так же, но если присутствует расширение (например, .txt или .jpg), метод от него избавляется. И наоборот, `GetExtension` возвращает расширение и ничего больше.

Если вам нужны временные файлы для выполнения какой-либо работы, `Path` предоставляет для этого три полезных метода. `GetRandomFileName` использует генератор случайных чисел для создания имени, которое можно

использовать для случайного файла или папки. Случайное число является криптографически сильным, что обеспечивает два полезных свойства: имя будет уникальным и труднопредсказуемым. (Определенные виды атак на систему безопасности могут стать возможными, если злоумышленник может предсказать имя или местоположение временных файлов.) Этот метод не создает ничего в файловой системе, а просто возвращает подходящее имя. `GetTempFileName`, с другой стороны, создаст файл в том месте, которое ОС предоставляет для временных файлов. Этот файл будет пустым, а метод вернет путь в виде строки. После этого вы можете открыть и изменить этот файл. (Этот способ не гарантирует использования криптографии для выбора действительно случайного имени, поэтому не стоит полагаться на такого рода местоположение файла, как имя, которое сложно угадать. Оно будет уникальным, но и только.) Вы должны удалить каждый файл, созданный `GetTempFileName`, как только закончите с ним работать. И наконец, `GetTempPath` возвращает путь к папке, которую будет использовать `GetTempFileName`; он ничего не создает, но вы можете использовать его в сочетании с именем, возвращаемым `GetRandomFileName` (объединив их с помощью `Path.Combine`), чтобы задать место, в котором нужно создать собственный временный файл.

FileInfo, DirectoryInfo и FileSystemInfo

Хотя классы `File` и `Folder` предоставляют вам доступ к информации, например о времени создания файла и о том, является ли он системным или только для чтения, с ними возникнет проблема, если необходим доступ к нескольким фрагментам информации. Не очень эффективно получать каждый кусочек данных с помощью отдельного вызова, так как эту информацию можно получить от базовой ОС за меньшее количество шагов. И иногда бывает проще передать один объект, содержащий все необходимые данные, вместо того чтобы искать место для размещения множества отдельных элементов. Как следствие пространство имён `System.IO` определяет классы `FileInfo` и `DirectoryInfo`, которые содержат информацию о файле или каталоге. Поскольку у них много общего, оба типа являются производными от базового класса `FileSystemInfo`.

Чтобы создать экземпляры этих классов, следует передать путь к нужному файлу или папке, как показано в листинге 15.16. Кстати, если через некоторое время вы решите, что файл мог быть изменен какой-то другой программой и захотите обновить информацию, которую возвращает `FileInfo`

или `DirectoryInfo`, вы можете вызвать `Refresh`, что обновит информацию, переданную файловой системой.

Листинг 15.16. Отображение информации о файле с помощью `FileInfo`

```
var fi = new FileInfo(@"c:\temp\log.txt");
Console.WriteLine(
    $"{fi.FullName} ({fi.Length} bytes) last modified on
    {fi.LastWriteTime}");
```

Помимо предоставления свойств различным методам `File` и `Directory`, которые извлекают информацию (`CreationTime`, `Attributes` и т. д.), эти информационные классы предоставляют методы экземпляров, которые соответствуют многим статическим методам `File` и `Directory`. Например, в `FileInfo` имеются методы `Delete`, `Encrypt` и `Decrypt`, работающие так же, как их тезки из `File`, за исключением того, что не нужно передавать аргумент пути. Существует также аналог `Move`, хотя и с другим именем — `MoveTo`.

`FileInfo` также содержит эквиваленты различных вспомогательных методов для открытия файла с помощью `Stream` или `FileStream`, таких как `AppendText`, `OpenRead` и `OpenText`. Что еще более удивительно, доступны также `Create` и `CreateText`. Оказывается, можно создать `FileInfo` для файла, которого еще не существует, а затем создать его с помощью этих вспомогательных методов. Он не пытается заполнить какие-либо свойства, описывающие файл, до тех пор пока вы впервые не попытаетесь прочитать их, и поэтому отложит вызов `FileNotFoundException` до этого момента в том случае, если вы создавали `FileInfo` для создания нового файла.

Как и следовало ожидать, `DirectoryInfo` также содержит методы экземпляров, которые соответствуют различным статическим вспомогательным методам, определенным в `Directory`.

Известные папки

Настольные приложения иногда должны использовать определенные папки. Например, настройки приложения обычно хранятся в определенной папке в профиле пользователя. Есть отдельная папка и для общесистемных настроек приложения. В Windows они обычно находятся в папке `AppData` пользователя и `C:\ProgramData` соответственно. Windows также определяет стандартные места для изображений, видео, музыки и документов. Кроме того, существуют папки, представляющие специальные функции оболочки, такие как рабочий стол и «избранное» пользователя.

Хотя эти папки в разных системах часто находятся в одном и том же месте, никогда не следует предполагать, что они всегда будут там, где вы ожидаете. (Поэтому в реальном коде вам никогда не следует делать то, что я делаю в листинге 15.15.) Многие из этих папок имеют отличные имена в локализованных версиях Windows. И даже в пределах конкретного языка нет никакой гарантии, что эти папки окажутся в обычном для них месте — некоторые из них можно перемещать, и их местоположения менялись в разных версиях Windows.

Таким образом, если нужен доступ к определенной стандартной папке, следует использовать метод `GetFolderPath` класса `Environment`, как показано в листинге 15.17. Метод принимает член вложенного типа перечисления `Environment.SpecialFolder`, который определяет значения для всех известных типов папок, доступных в Windows.

Листинг 15.17. Поиск места для сохранения настроек

```
string appSettingsRoot =
    Environment.GetFolderPath(
        Environment.SpecialFolder.ApplicationData);
string myAppSettingsFolder =
    Path.Combine(appSettingsRoot, @"Endjin\FrobnicatorPro");
```



В системах, отличных от Windows, `GetFolderPath` возвращает пустую строку для большинства записей перечисления, потому что у них нет локального эквивалента. Тем не менее часть из них работают, например `MyDocuments`, `CommonApplicationData` и `UserProfile`.

Папка `ApplicationData` находится в разделе `roaming` профиля пользователя. Информация, которую не нужно копировать на другие используемые машины (например, кэш, который может быть восстановлен при необходимости), должна находиться в локальном разделе, который можно получить с помощью записи `enum LocalApplicationData`.

Сериализация

Типы `Stream`, `TextReader` и `TextWriter` предоставляют возможность записи и чтения данных из файлов, сети или чего-то похожего на поток, что представляет подходящий конкретный класс. Но данные абстракции поддер-

живают только байтовые или текстовые данные. Предположим, у вас есть объект с несколькими свойствами различного типа, включая числовые типы и, возможно, ссылки на другие объекты, некоторые из которых могут быть коллекциями. Что делать, если необходимо записать всю информацию, содержащуюся в этом объекте, в файл или передать по сетевому соединению, чтобы объект того же типа и с такими же значениями свойств можно было восстановить позднее или на другом компьютере на другом конце сетевого соединения?

Это можно проделать с помощью абстракций, показанных в этой главе, но для этого потребуется значительный объем работы Вам придется написать код для чтения каждого свойства и записать его значение в `Stream` или `TextWriter`, а также преобразовать его в двоичный или текстовый формат. Вам также придется определиться с вашим представлением: вы просто запишете значения в фиксированном порядке или придумаете схему для записи пар имя/значение, чтобы формат вас не ограничивал, когда позднее понадобится добавить новые свойства? Кроме того, вам придется найти способы обработки коллекций и ссылок на другие объекты, а также решить, что делать с циклическими ссылками, когда два объекта ссылаются друг на друга (на чем простой код может зациклиться).

.NET предлагает несколько решений этой проблемы, и каждый из них подразумевает различные компромиссы между сложностью поддерживаемых сценариев, которые они могут поддерживать, тем, насколько хорошо они справляются с управлением версиями, и тем, насколько они подходят для взаимодействия с другими платформами. Все эти методы подпадают под широкую категорию под названием «серIALIZАЦИЯ» (потому что включают в себя запись состояния объекта в некую форму, которая хранит данные последовательно, например в `Stream`). За многие годы в .NET было внедрено много разных механизмов, поэтому я не буду описывать их все. Я просто представлю те из них, которые лучше всего реализуют конкретные подходы к проблеме.

BinaryReader, BinaryWriter и BinaryPrimitives

Хотя они не являются формами сериализации в строгом смысле слова, никакое обсуждение из этой области не обойдется без охвата классов `BinaryReader` и `BinaryWriter`. Причина в том, что они решают фундаментальную проблему, с которой сталкивается любая попытка сериализации и десериализации

объектов: они способны преобразовать внутренние типы CLR в поток байтов и обратно. `BinaryPrimitives` делает то же самое, но может работать со `Span<byte>` и родственными типами, которые обсуждаются в главе 18.

`BinaryWriter` — это обертка для записываемого `Stream`. Она предоставляет метод `Write`, который имеет перегрузки для всех внутренних типов, кроме `object`. Таким образом, этот класс может принимать значение любого из числовых типов, а также типов `string`, `char` или `bool`, после чего записывает двоичное представление этого значения в `Stream`. Он также может записывать массивы типа `byte` или `char`.

`BinaryReader` — это обертка для читаемого `Stream`, которая предоставляет различные методы для чтения данных, каждый из которых соответствует перегрузкам `Write`, предоставляемым `BinaryWriter`. Например, он содержит `ReadDouble`, `ReadInt32` и `ReadString`.

Чтобы использовать эти типы при сериализации данных, необходимо создать `BinaryWriter` и передать ему каждое значение, которое вы хотите сохранить. Когда позже вам понадобится десериализовать эти данные, вы обернете содержащий данные поток в `BinaryReader` и вызовете соответствующие методы чтения в том же порядке, в котором вы изначально записывали данные.

`BinaryPrimitives` работает несколько иначе. Он разработан для кода, которому требуется минимизировать количество выделений кучи, и поэтому не является типом-оберткой. Это статический класс, содержащий широкий спектр методов, таких как `ReadInt32LittleEndian` и `WriteUInt16BigEndian`.

Они принимают аргументы `ReadOnlySpan<byte>` и `Span<byte>` соответственно, так как тип предназначен для работы непосредственно с данными, где бы они ни располагались в памяти (не обязательно в виде `Stream`). Однако основной принцип тот же: он выполняет преобразование между последовательностями байтов и простыми типами .NET. (Кроме того, обработка строк здесь более сложная: отсутствует метод `ReadString`, потому что все, что возвращает строку, приведет к созданию нового строкового объекта в куче, если только у вас нет фиксированного набора возможных строк, которые вы можете предварительно выделить и раз за разом передавать. Подробности ищите в главе 18.)

Эти классы решают только одну проблему, а именно представление различных типов .NET в двоичном виде. Вам все еще предстоит определиться, каким образом представлять целые объекты и что делать с более сложными видами структур, такими как ссылки между объектами.

Сериализация CLR

Как следует из названия, сериализация CLR — это функция, встроенная в саму среду выполнения, а не просто часть функционала библиотеки. Она не поддерживалась в .NET Core первые несколько версий, но в итоге Microsoft вернула ее в несколько сокращенном виде, чтобы упростить миграцию приложений из .NET Framework. Microsoft старается не поощрять ее использование, но функция продолжает широко использоваться в определенных сценариях. Например, ее используют в микросервисных средах для отправки исключений и относительно простых структур данных, которые пересекают границы служб. Ограниченнная поддержка в .NET Core нацелена на эти сценарии, поэтому вы не можете сериализовать таким способом какой-либо старый объект .NET.

Наиболее интересным аспектом сериализации CLR является то, что она имеет дело непосредственно со ссылками на объекты. Если вы сериализуете, скажем, `List<SomeType>`, где несколько записей в списке ссылаются на один и тот же объект, то сериализация CLR обнаружит это и сохранит только одну копию объекта, а при десериализации восстановит эту структуру «один объект — много ссылок». (Системы сериализации, основанные на широко используемом формате JSON, обычно этого не делают.)

Типы должны подключать сериализацию CLR самостоятельно. .NET определяет атрибут `[Serializable]`, который должен присутствовать на момент сериализации вашего типа CLR. Но как только вы его добавите, CLR позаботиться обо всех остальных деталях процесса. В листинге 15.18 показан тип с этим атрибутом, который я и использую для демонстрации процесса сериализации.

Листинг 15.18. Сериализуемый тип

```
using System;
using System.Collections.Generic;
using System.Linq;

[Serializable]
class Person
{
    public string Name { get; set; }

    public IList<Person> Friends { get; } = new List<Person>();

    public override string ToString() =>
        $"{Name} (friends: {string.Join(", ", Friends.Select(f =>
            f.Name))})";
}
```

Сериализация работает напрямую с полями объекта. Она использует отражение, что позволяет получить доступ ко всем членам, будь они `public` или `private`. В этом примере класса имеются два скрытых и сгенерированных компилятором поля для свойств `Friends` и `Name`. (`List<T>`, кстати, тоже имеет атрибут `[Serializable]`. Если бы это было не так, пример бы не заработал.) Как показано в листинге 15.19, мы можем использовать тип `BinaryFormatter` (в пространстве имен `System.Runtime.Serialization.Formatters.Binary`) для сериализации экземпляра этого типа в поток.

Листинг 15.19. Сериализация с помощью `BinaryFormatter`

```
var stream = new MemoryStream();
var serializer = new BinaryFormatter();
serializer.Serialize(stream, person);
```

Как показано в листинге 15.20, тип `BinaryFormatter` также выполняет и десериализацию.

Листинг 15.20. Десериализация с помощью `BinaryFormatter`

```
stream.Seek(0, SeekOrigin.Begin);
var serializer = new BinaryFormatter();
var personCopy = (Person) serializer.Deserialize(stream);
```

Если переменная `person` в листинге 15.19 ссылается на объект, свойство `Friends` которого возвращает коллекцию, содержащую ссылки на объекты `Person` со свойствами `Friends`, которые содержат коллекции, которые, в свою очередь, ссылаются на тот же объект, что и `person`, то это будет означать, что у нас имеются циклические ссылки. `BinaryFormatter` это обнаруживает, сохраняя в потоке лишь одну копию каждого объекта, а при десериализации правильно восстанавливает любую такую структуру.

Это довольно полезно, когда, просто добавив один атрибут, можно записать полное дерево объектов. Но есть и обратная сторона: если я изменю реализацию любого из сериализуемых типов, у меня возникнут проблемы, когда новая версия моего кода попытается десериализовать поток, созданный старой версией. Так что это не лучший выбор для записи настроек приложения на диск, потому что они могут изменяться с каждой новой версией. Когда подобное происходит, можно настроить способ работы сериализации, позволяющий поддерживать управление версиями, но в этот момент мы возвращаемся к выполнению большого количества работы вручную. (На самом деле, может быть, проще использовать `BinaryReader` и `BinaryWriter`.) Также

легко представить, какие проблемы безопасности вызывает такой стиль сериализации: тот, кто контролирует поток, который вы десериализуете, фактически имеет полный контроль над всеми полями ваших объектов.

Другая проблема с сериализацией CLR состоит в том, что она создает двоичные потоки в формате .NET. Если с потоком работает код только на .NET, то это не проблема, но потоки можно создавать и в расчете на более широкую аудиторию. Есть и другие механизмы сериализации кроме сериализации CLR, и они способны создавать потоки, которые другим системам использовать гораздо проще.

JSON.NET

Удивительно, но наиболее широко используемый в .NET механизм сериализации написан не Microsoft, хотя некоторые библиотеки Microsoft его используют. Библиотека JSON.NET – это проект с открытым исходным кодом, написанный Джеймсом Ньютоном-Кингом и выпущенный под лицензией MIT. Вы можете найти его на <http://www.newtonsoft.com/json> или через NuGet под именем `Newtonsoft.Json`. Как следует из названия, он работает с JSON (*JavaScript Object Notation*) – чрезвычайно популярным форматом обмена данными. Производительность, относительная простота использования и всесторонняя поддержка всех разновидностей .NET сделали его ходовой библиотекой для сериализации JSON.



Одно время платформа построения веб-приложений ASP.NET Core внутри использовала JSON.NET. Однако в .NET Core 3.0 и .NET Standard 2.1 появилась новая библиотека JSON, созданная Microsoft и состоящая из различных типов в пространстве имен `System.Text.Json`. Они используют новые, экономные в плане потребления памяти методы, описанные в главе 18, что делает их несколько менее удобными, но более эффективными. Ядро ASP.NET перешло на эту библиотеку, частично для скорости, но также и для того, чтобы избавиться от зависимости. (Есть несколько разных версий JSON.NET, и могут возникнуть конфликты зависимостей, если среда зависит от одной конкретной версии, а вы хотите использовать библиотеку, которая зависит от другой версии.)

JSON.NET поддерживает три способа работы с данными JSON. Она определяет интерфейсы `JsonReader` и `JsonWriter`, которые являются потоковыми абстракциями и представляют содержимое данных JSON в виде после-

довательности элементов. Они могут пригодиться, если вам требуется обрабатывать документы JSON, которые слишком велики для загрузки в память в виде единого объекта, но чаще вы будете использовать эти типы в качестве средства получения и передачи данных в другие механизмы JSON.NET. На практике, как правило, проще использовать `JsonSerializer` (он часто используется косвенно через более простой вспомогательный класс `JsonConvert`). Он выполняет преобразование между объектами и целыми потоками JSON. Он требует определения классов со структурой, соответствующей JSON. Наконец, есть более динамичный вариант под названием LINQ to JSON. Как следует из названия, он поддерживает запросы LINQ к данным JSON, но это не все: он пригодится, когда во время разработки вы точно не знаете, какой будет структура ваших данных JSON.

JsonSerializer и JsonConvert

`JsonSerializer` предлагает модель сериализации на основе атрибутов, в которой вы определяете один или несколько классов, отражающих структуру данных JSON, с которой вам нужно работать, после чего можете преобразовывать данные JSON в эту модель и из нее. Как правило, вы не будете использовать тип `JsonSerializer` напрямую. Если вам не нужен точный контроль над рядом аспектов сериализации, вам хватит вспомогательного `JsonConvert`.

В листинге 15.21 показана простая модель, подходящая для использования с JSON.NET. Как видите, я не обязан использовать какой-либо конкретный базовый класс или какие-либо обязательные атрибуты.

Листинг 15.21. Простая модель JSON.NET

```
public class SimpleData
{
    public int Id { get; set; }
    public IList<string> Names { get; set; }
    public NestedData Location { get; set; }
    public IDictionary<string, int> Map { get; set; }
}

public class NestedData
{
    public string LocationName { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}
```

В листинге 15.22 создается экземпляр этой модели, а затем используется метод `SerializeObject` класса `JsonConvert` для сериализации его в строку. (Под капотом здесь используется `JsonSerializer`. Для простых сценариев проще всего использовать `JsonConvert`, потому что гибкость `JsonSerializer` лишь все усложняет.)

Листинг 15.22. Сериализация данных с помощью `JsonConvert`

```
var model = new SimpleData
{
    Id = 42,
    Names = new[] { "Bell", "Stacey", "her", "Jane" },
    Location = new NestedData
    {
        LocationName = "London",
        Latitude = 51.503209,
        Longitude = -0.119145
    },
    Map = new Dictionary<string, int>
    {
        { "Answer", 42 },
        { "FirstPrime", 2 }
    }
};

string json = JsonConvert.SerializeObject(model, Formatting.Indented);
Console.WriteLine(json);
```

Второй аргумент для `SerializeObject` является необязательным. Я использовал его для отступа в JSON, чтобы его было легче читать. (По умолчанию JSON.NET будет использовать более эффективный макет без лишних пробелов, но его будет гораздо сложнее читать.) Результаты выглядят так:

```
{
    "Id": 42,
    "Names": [
        "Bell",
        "Stacey",
        "her",
        "Jane"
    ],
    "Location": {
        "LocationName": "London",
```

```
        "Latitude": 51.503209,  
        "Longitude": -0.119145  
    },  
    "Map": {  
        "Answer": 42,  
        "FirstPrime": 2  
    }  
}
```

Как видите, каждый объект стал объектом JSON, в котором пары имя/значение соответствуют свойствам в моей модели. Числа и строки представлены именно так, как вы ожидаете. `IList<string>` стал массивом JSON, а `IDictionary<string, int>` стал еще одним словарем JSON. Я использовал интерфейсы для этих коллекций, но вы можете использовать и конкретные типы `List<T>` и `Dictionary< TKey, TValue >`. Если хотите, то для представления списков можно использовать обычные массивы. Мне больше нравятся интерфейсы, потому что они позволяют более свободно использовать любые требуемые типы коллекций. (Например, в листинге 15.22 использовался строковый массив, но можно было бы использовать и `List<string>` без изменения типа модели.)

Как показано в листинге 15.23, преобразование сериализованного JSON обратно в модель не менее просто.

Листинг 15.23. Десериализация данных с помощью `JsonConvert`

```
var deserialized = JsonConvert.DeserializeObject<SimpleData>(json);
```

Хотя такой простой и понятной модели уже может быть достаточно, иногда требуется взять под контроль некоторые аспекты сериализации, особенно если вы работаете с внешним форматом JSON. Например, некоторые API используют соглашения касательно регистра, которые отличаются от .NET. Например, довольно популярная `camelCasing` противоречит соглашению `PascalCasing` для свойств .NET. Проблему можно решить, использовав атрибут `JsonProperty` для указания имени для использования в JSON. Обратимся к листингу 15.24.

При сериализации JSON.NET будет использовать имена, указанные в `JsonProperty`, и их же библиотека будет искать при десериализации. В качестве альтернативы вы можете сообщить JSON.NET, что хотите использовать данное соглашение о регистре для всех свойств, передав соответствующим

образом сконфигурированные `JsonSerializerSettings` в `JsonConvert`. В таком случае вам эти атрибуты не понадобятся. Есть возможность осуществлять и более глубокий контроль, например определив собственные механизмы сериализации типов данных. (Например, вам может потребоваться представить что-то в вашем коде C# в виде `DateTimeOffset`, но так, чтобы в JSON это стало строкой с определенным форматом даты и времени.) Полную информацию можно найти в документации по JSON.NET.

Листинг 15.24. Управление JSON с помощью атрибутов `JsonProperty`

```
public class NestedData
{
    [JsonProperty("locationName")]
    public string LocationName { get; set; }

    [JsonProperty("latitude")]
    public double Latitude { get; set; }

    [JsonProperty("longitude")]
    public double Longitude { get; set; }
}
```

LINQ в JSON

В то время как `JsonSerializer` требует, чтобы вы определили один или несколько типов для представления структуры JSON, с которой вы хотите работать, JSON.NET содержит набор типов, которые используют более динамичный подход, называемый LINQ to JSON. При этом происходит разбор данных JSON и преобразование их в объекты типа `JObject`, `JArray`, `JProperty` и `JValue`, все из которых являются производными от базового класса `JToken`. Использование этих типов аналогично работе с JSON из JavaScript — вы просто можете получить прямой доступ к содержимому, не определяя классы (недостаток состоит в том, что определенные виды ошибок, которые компилятор обнаружит с помощью подхода `JsonSerializer`, в данном случае обнаружатся только на этапе выполнения). В листинге 15.25 этот метод используется для чтения части данных из того же JSON, который использовался в последних нескольких примерах.

Как видите, `JObject` предоставляет индексатор, с помощью которого вы можете получать свойства объектов JSON. Для перебора массива имен мне удалось использовать цикл `foreach`. Это сработает и в случае вложенных объектов, таких как объект в свойстве `Location`. Поскольку данный API

ничего не знает о конкретных типах, все определяется в терминах базового типа `JToken`, причем конкретный тип определяется тем, что JSON.NET обнаруживает во время выполнения. Таким образом, в случае данных, которые у нас есть, первый цикл `foreach` обнаружит серию объектов `JValue` (по одному для каждой строки в массиве), а второй — серию объектов `JProperty` (по одному для каждого свойства во вложенном объекте).

Листинг 15.25. Чтение JSON с помощью `JToken`

```
var jo = (JObject) JToken.Parse(json);
Console.WriteLine(jo["Id"]);
foreach (JToken name in jo["Names"])
{
    Console.WriteLine(name);
}
foreach (JToken loc in jo["Location"])
{
    Console.WriteLine(loc);
}
```

В листинге 15.25 я немного схитрил, передав каждый `JToken` в `Console.WriteLine`. Каждый конкретный тип, производный от `JToken`, реализует `ToString`, так что данная программа выдаст достаточно адекватный вывод:

```
42
Bell
Stacey
her
Jane
"locationName": "London"
"latitude": 51.503209
"longitude": -0.119145
```

Но что, если вам потребуется работать с данными, а не просто показывать их пользователю? С массивами и вложенными объектами можно работать, приведя каждый `JToken` к типу, которым он, по-вашему, является (либо потому, что вы проверили тип во время выполнения, либо просто потому, что у вас есть основания полагать, что данные будут в конкретном формате). Для значений можно использовать метод извлечения данных `JToken<T>` класса `JToken`, задав тип, который, по вашему мнению, должно иметь значение в качестве аргумента обобщенного типа. Реализацию можно увидеть в листинге 15.26.

Листинг 15.26. Работа с данными в LINQ to JSON

```
int id = jo["Id"].Value<int>();
var names = (JArray) jo["Names"];
string firstName = names[0].Value<string>();
```

Каждая строка этого примера способна выдать исключение во время выполнения. Если свойство `Id` отсутствует или его невозможно преобразовать в `int`, первая строка завершится ошибкой. Если свойство `Names` отсутствует или не содержит массив, вторая строка завершится ошибкой. И если массив пуст или его первый элемент не может быть преобразован в строку (например, потому что он вложенный), последняя строка тоже приведет к ошибке.

Плюс в том, что вам не нужно определять какие-либо типы для соблюдения модели данных. Кроме того, гораздо проще писать код, поведение которого определяется структурой данных, потому что данный API способен описать то, что обнаружил. Например, вы уже видели, что использование `foreach` в `JObject` создает последовательность объектов `JProperty`. Это можно использовать для написания запросов к JSON, откуда данный API и получил свое имя. Листинг 15.27 находит все элементы `JProperty` в данных, где первая буква имени свойства строчная.

Листинг 15.27. Запрос к данным JSON

```
IEnumerable<JProperty> propsStartingWithLowerCase = jo.Descendants()
    .OfType<JProperty>()
    .Where(p => char.ToLower(p.Name[0]));
foreach (JProperty p in propsStartingWithLowerCase)
{
    Console.WriteLine(p);
}
```

Методы `OfType` и `Where` в данном случае взяты из LINQ to Objects. Библиотека JSON.NET не предоставляет собственной реализации стандартных операторов LINQ. Он поддерживает LINQ тем, что просто представляет структуру ваших данных JSON посредством реализации `IEnumerable<T>`, что позволяет использовать LINQ to Objects для выполнения запросов. Единственное, что он добавляет и что может оказаться полезным, — это методы вроде `Descendants`, который рекурсивно обходит всю структуру JSON ниже определенного узла, возвращая каждый `JToken` в виде одной плоской коллекции.

Итог

Класс `Stream` — это абстракция, представляющая данные в виде последовательности байтов. Поток может поддерживать чтение, запись или и то и другое, а также поиск по произвольному смещению или прямой последовательный доступ. `TextReader` и `TextWriter` обеспечивают строго последовательное чтение и запись символьных данных, абстрагируясь от кодировки символов. Эти типы могут работать поверх файла, сетевого подключения или памяти. Кроме того, вы можете реализовать собственные версии этих абстрактных классов. Класс `FileStream`, в свою очередь, предоставляет ряд других функций доступа к файловой системе, но для полного контроля у нас также есть классы `File` и `Directory`. Когда байтов и строк недостаточно, можно использовать различные механизмы сериализации .NET, которые способны автоматизировать преобразование между состоянием объекта в памяти и представлением, которое можно записать на диск, отправить по сети или передать в любой другой поток; позднее это представление можно превратить в объект того же типа с эквивалентным состоянием.

Многопоточность

Многопоточность позволяет приложению выполнять несколько частей кода одновременно. Есть две распространенные причины на использование многопоточности. Одна из них — это использование возможностей параллельной обработки данных. Многоядерные процессоры сейчас используются более или менее повсеместно, и, чтобы полностью реализовать их потенциал производительности, нужно предоставить ЦП несколько потоков работы, чтобы каждое ядро было занято чем-то полезным. Другая обычная причина написания многопоточного кода — предотвращение остановки процесса при выполнении медленных задач, например при чтении с диска.

Многопоточность — не единственный способ решить эту вторую проблему, так как асинхронные методы могут оказаться даже предпочтительнее. C# содержит функционал поддержки асинхронной работы. Асинхронное выполнение необязательно означает многопоточность, но на практике связь часто присутствует, так что в этой главе я опишу некоторые модели асинхронного программирования. Тем не менее эта глава посвящена основам работы с потоками. Поддержку асинхронного кода на уровне языка я опишу в главе 17.

Потоки

Все операционные системы, в которых работает .NET, позволяют каждому процессу содержать несколько потоков. Каждый поток имеет собственный стек, и ОС создает иллюзию того, что поток получает в свое пользование весь *аппаратный поток* ЦП. (См. врезку «Процессоры, ядра и аппаратные потоки».) В ОС можно создать гораздо больше потоков, чем количество аппаратных потоков, предоставляемых вашим компьютером, поскольку ОС виртуализирует ЦП, переключая контекст из одного потока в другой. Компьютер, который я использую при написании этого текста, имеет 16 аппаратных потоков, что достаточно много, но это не то же самое, что 8893 потока, которые в настоящее время активны в различных процессах на этой же машине.

ПРОЦЕССОРЫ, ЯДРА И АППАРАТНЫЕ ПОТОКИ

Аппаратный поток — это часть оборудования, способная выполнять код. Еще в начале 2000-х один процессорный чип давал вам один аппаратный поток, а несколько аппаратных потоков можно было получить только на компьютерах с несколькими физически отдельными процессорами, подключенными к отдельным разъемам на материнской плате. Однако два изобретения усложнили отношения между оборудованием и потоками: это многоядерные процессоры и гиперпоточность.

В случае многоядерного процессора вы по сути получаете несколько процессоров на одном кусочке кремния. Если вскрыть компьютер и подсчитать количество процессорных чипов, то вы не обязательно узнаете, сколько аппаратных потоков у вас в распоряжении. Но если бы у вас была возможность рассмотреть кремний процессора с помощью подходящего микроскопа, вы бы увидели на чипе, рядом друг с другом, два или более отдельных процессора.

Гиперпоточность, также называемая *одновременной многопоточностью* (simultaneous multithreading, SMT), усложняет ситуацию еще больше. Гиперпоточное ядро — это один процессор, имеющий два набора определенных частей. (Может быть и больше, но удвоение наиболее распространено.) Таким образом, даже если выполнять деление с плавающей запятой способна лишь одна часть ядра, фактически у нас будет два набора регистров. Каждый набор регистров включает в себя регистр указателя инструкций (IP), который отслеживает, до какой точки дошло выполнение. Регистры также содержат непосредственное рабочее состояние кода, поэтому, имея два набора, одно ядро может выполнять код из двух мест одновременно — другими словами, гиперпоточность позволяет одному ядру предоставлять два аппаратных потока. Поскольку удваиваются только определенные части ЦП, некоторые ресурсы двух контекста выполнения должны использовать совместно — они не могут одновременно выполнять операции деления с плавающей запятой, потому что для этого в ядре есть только один экземпляр оборудования. Но если одному из аппаратных потоков требуется выполнить деление, в то время как другому перемножить два числа, они, как правило, будут делать это параллельно, поскольку данные операции выполняются различными областями ядра. Гиперпоточность обеспечивает одновременную работу еще большего количества частей одного ядра процессора. Она не дает такой же пропускной способности, как два полноценных ядра (потому что если оба аппаратных потока хотят одновременно выполнять одну и ту же работу, одному из них придется ожидать), но часто способна обеспечить лучшую пропускную способность в случае каждого ядра, чем это было бы возможно в ином случае.

В гиперпоточной системе общее количество доступных аппаратных потоков — это количество ядер, умноженное на количество модулей с многопоточным исполнением в каждом ядре. Например, процессор Intel Core i9-9900K имеет 8 ядер с двусторонней гиперпоточностью, что дает в общей сложности 16 аппаратных потоков.

CLR предоставляет собственную абстракцию потоков поверх потоков ОС. Как правило, связь в данном случае прямая — если вы пишете консольное приложение, настольное приложение Windows или веб-приложение, каждый объект .NET Thread напрямую соответствует конкретному базовому потоку ОС. Однако нет гарантий, что такое отношение будет существовать, — среда CLR была разработана, чтобы позволить потоку .NET переключаться между различными потоками ОС. Это возможно только в приложении, которое использует API неуправляемого хостинга CLR для регулировки отношений между CLR и содержащим его процессом. (Например, так работает функция интеграции CLR в SQL Server.) В большинстве случаев на практике поток CLR будет соответствовать потоку ОС, но код библиотеки должен по возможности не зависеть от этого; код, который руководствуется таким предположением, может перестать работать при использовании в приложении, предоставляющем собственное хост-приложение CLR.

Вскоре я вернусь к классу `Thread`, но прежде, чем писать многопоточный код, нужно разобраться с основными правилами управления состоянием при использовании нескольких потоков¹.

Потоки, переменные и совместно используемое состояние

Каждый поток CLR получает различные относящиеся к потоку ресурсы, такие как стек вызовов (который содержит аргументы метода и некоторые локальные переменные). Поскольку каждый поток обладает собственным стеком, локальные переменные, которые в итоге в нем окажутся, будут для этого потока локальными. Каждый раз при вызове метода вы получаете новый набор его локальных переменных. Это не только лежит в основе рекурсии, но также важно и в многопоточном коде, потому что данные, которые доступны нескольким потокам, требуют при работе гораздо большей осторожности, особенно если они изменяются. Координация доступа к общим данным — сложная задача. Кое-какие приемы я опишу в разделе «Синхронизация» на с. 828, но лучше по возможности избегать этой необходимости, и локальная природа стека в пределах потока может в данном случае очень пригодиться.

Представим веб-приложение. Нагруженные сайты должны обрабатывать запросы от нескольких пользователей одновременно, так что вы, скорее

¹ В этой главе я много раз использовал слово «состояние», под которым я просто имею в виду информацию, хранящуюся в переменных и объектах.

всего, окажетесь в ситуации, когда определенный фрагмент кода (например, код домашней страницы вашего сайта) выполняется одновременно в нескольких разных потоках — ASP.NET Core использует многопоточность, чтобы иметь возможность предоставить одну и ту же логическую страницу нескольким пользователям. (Поскольку страницы часто адаптируются под конкретных пользователей, веб-сайты обычно не просто выдают один и тот же контент, поэтому если 1000 пользователей захотят увидеть домашнюю страницу, будет 1000 раз запущен код, который генерирует эту страницу.) ASP.NET Core предоставляет вам различные объекты, которые могут понадобиться вашему коду, но большинство из них относятся к конкретному запросу. Таким образом, если ваш код может работать только с этими объектами и с локальными переменными, каждый поток будет способен работать полностью независимо. Если требуется совместно используемое состояние (например, объекты, которые видны нескольким потокам, скажем, через статическое поле или свойство), ваша жизнь может усложниться. Однако обычные локальные переменные, как правило, незамысловаты.

Почему только «как правило»? Все усложняется, если вы используете лямбда-выражения или анонимные функции, которые позволяют объявить переменную в содержащем методе, а затем использовать во внутреннем. Эта переменная становится доступна двум или более методам, и в случае многопоточности вполне возможно, что эти методы могут выполняться одновременно. (Если речь идет о CLR, то это уже не локальная переменная — это поле в классе, сгенерированном компилятором.) Совместное использование локальных переменных в нескольких методах снимает гарантию полной локальности, поэтому приходится работать с такими переменными так, как вы бы работали с более очевидными общими элементами — статическими свойствами и полями.

Еще один важный момент, о котором следует помнить в многопоточной среде, — это разница между переменной и объектом, на которую она ссылается. (Такая проблема возникает только с переменными ссылочного типа.) Хотя локальная переменная доступна только внутри своего метода объявления, она может быть не единственной переменной, которая ссылается на конкретный объект. Иногда так и происходит, но если вы создаете объект внутри метода и не сохраняете его куда-либо еще (что делает его доступным для более широкой аудитории), то вам не о чем беспокоиться. `StringBuilder` из листинга 16.1 всегда используется только в методе, который его создает.

Листинг 16.1. Видимость объекта и методы

```
public static string FormatDictionary<TKey, TValue>(  
    IDictionary<TKey, TValue> input)  
{  
    var sb = new StringBuilder();  
    foreach (var item in input)  
    {  
        sb.AppendFormat("{0}: {1}", item.Key, item.Value);  
        sb.AppendLine();  
    }  
  
    return sb.ToString();  
}
```

Здесь не нужно беспокоиться о том, могут ли другие потоки попытаться изменить `StringBuilder`. Нет тут и вложенных методов, поэтому переменная `sb` на самом деле локальная и единственная содержит ссылку на `StringBuilder`. (Мы полагаемся на тот факт, что `StringBuilder` не может скрытно хранить копии этой ссылки там, где их могут найти другие потоки.)

Но как насчет аргумента `input`? Он тоже локальный в рамках метода, но объект, на который он ссылается, таковым не является: код, вызывающий `FormatDictionary`, самостоятельно принимает решение о том, на что ссылается `input`. Глядя лишь на листинг 16.1, невозможно сказать, используется ли объект словаря, на который он ссылается, другими потоками. Вызывающий код может создать один словарь, а затем создать два потока, и один из них может изменить словарь, в то время как другой вызывает метод `FormatDictionary`. В результате может возникнуть проблема: большинство реализаций словаря не поддерживают изменение в одном потоке одновременно с использованием в другом потоке. И даже если вы работаете с коллекцией, которая разработана для подобного одновременного использования, вам часто не будет разрешено изменять ее во время перечисления ее содержимого (например, в цикле `foreach`).

Вы можете решить, что любая коллекция, предназначенная для одновременного использования из нескольких потоков (*потокобезопасная коллекция*), должна позволять одному потоку перебирать ее содержимое, а другому — изменять его. Если это запрещено, то в каком смысле она потокобезопасна? Фактически основным отличием между потокобезопасной и обычной коллекцией при данном сценарии является предсказуемость: тогда как потокобезопасная коллекция может выдать исключение, когда обнаружит, что сценарий сработал, то в случае обычной коллекции нет никаких гаран-

тий, что она как-то на это отреагирует. Подобное развитие событий может привести к сбою или исказить результаты итерации, в результате чего одна запись будет фигурировать в результатах несколько раз. Может произойти что угодно, потому что она используется способом, который не поддерживается. Иногда потокобезопасность означает лишь то, что сбой происходит в предсказуемом месте в предсказуемое время.

Так повелось, что различные коллекции в пространстве имен `System.Collections.Concurrent` по факту поддерживают изменения во время перечисления, не вызывая исключений. Но они по большей части имеют специальные API для поддержки параллельных вычислений, что отличает их от других классов коллекций, и поэтому не всегда являются адекватной заменой.

Листинг 16.1 никак не может обеспечить безопасное использование своего аргумента `input` в многопоточных средах, потому что это отдано на откуп вызывающей стороне. Проблемы параллелизма нужно решать на более высоком уровне. Термин «потокобезопасный» способен ввести в заблуждение, потому что предполагает нечто в общем случае невозможное. Неопытные разработчики часто попадают в ловушку, думая, что с них снимается всякая ответственность за проблемы с многопоточностью в коде просто в силу того, что все используемые ими объекты потокобезопасны. Но это может не сработать, потому что хотя отдельные потокобезопасные объекты будут стремиться поддерживать собственную целостность, нет гарантий, что состояние вашего приложения в целом будет согласованным.

Чтобы проиллюстрировать это, в листинге 16.2 использован класс `ConcurrentDictionary< TKey, TValue >` из пространства имен `System.Collections.Concurrent`. Каждая определенная в нем операция является потоконезависимой в том смысле, что поддерживает объект в согласованном состоянии и дает ожидаемый результат с учетом состояния коллекции до вызова. Однако этот пример умудряется использовать его способом, который не является потокобезопасным.

Код выглядит вполне безопасным. (А еще он выглядит бессмысленным; он предназначен лишь для того, чтобы показать, как даже в очень простом коде что-то может пойти не так.) Но если экземпляр словаря используется несколькими потоками (что вероятно, учитывая выбор нами типа, разработанного специально для многопоточного использования), то вполне возможно, что в промежутке между установкой значения для ключа 1 и попыткой его получения какой-то другой поток удалит эту запись. Если я помешу этот

код в программу, которая многократно запускает данный метод в нескольких потоках, но в которой также есть несколько других потоков, удаляющих ту же самую запись, в итоге меня ждет исключение `KeyNotFoundException`.

Листинг 16.2. Непотокобезопасное использование потокобезопасной коллекции

```
static string UseDictionary(ConcurrentDictionary<int, string> cd)
{
    cd[1] = "One";
    return cd[1];
}
```

Для обеспечения согласованности всей системы параллельным системам требуется стратегия «сверху вниз». (Вот почему системы управления базами данных часто используют транзакции, которые группируют наборы операций вместе в виде атомарных единиц работы, которые либо полностью успешно выполняются, либо не имеют никакого эффекта. Эта атомарная группировка является важной частью механизма, с помощью которого транзакции помогают сохранить согласованность состояния в масштабе всей системы.) В случае листинга 16.1 это означает, что ответственность за свободное использование словаря во время выполнения метода лежит на коде, который вызывает `FormatDictionary`.



Хотя вызывающий код должен гарантировать, что любые переданные им объекты безопасны для использования во время вызова метода, в целом не следует полагаться на то, что можно сохранить на будущее ссылки на ваши аргументы. В случае анонимных функций и делегатов это легко сделать случайно — если вложенный метод ссылается на аргументы содержащего его метода и при этом запускается после завершения содержащего метода, нет оснований полагать, что вам будет разрешен доступ к объектам, на которые ссылаются аргументы. Если вам все же это требуется, то стоит задокументировать свои предположения о том, когда вы можете использовать объекты, и проверять любой код, который вызывает метод, стараясь в каждом случае убедиться, что эти предположения по-прежнему верны.

Локальное хранилище потока

Иногда бывает полезно поддерживать локальное состояние потока более широко, нежели в пределах одного метода. Различные части библиотеки классов .NET так и делают. Например, пространство имен `System`.

`Transactions` определяет API для использования транзакций с базами данных, очередями сообщений и любыми другими менеджерами ресурсов, которые их поддерживают. Он предоставляет неявную модель, в которой вы можете запустить внешнюю транзакцию, а любые поддерживающие это операции будут включены в нее без необходимости передавать какие-либо явные аргументы, связанные с транзакцией. (Он также поддерживает явную модель, если вы предпочитаете ее.) Статическое свойство `Current` класса `Transaction` возвращает внешнюю транзакцию для текущего потока или `null`, если у потока в процессе в данный момент нет внешней транзакции.

Для поддержки такого рода поточного состояния .NET предлагает класс `ThreadLocal<T>`. Листинг 16.3 использует это для предоставления обертки делегата, которая позволяет лишь одному вызову делегата выполняться в любом потоке в любое время.

Листинг 16.3. Использование `ThreadLocal<T>`

```
class Notifier
{
    private readonly ThreadLocal<bool> _isCallbackInProgress =
        new ThreadLocal<bool>();

    private readonly Action _callback;

    public Notifier(Action callback)
    {
        _callback = callback;
    }

    public void Notify()
    {
        if (_isCallbackInProgress.Value)
        {
            throw new InvalidOperationException(
                "Notification already in progress on this thread");
        }

        try
        {
            _isCallbackInProgress.Value = true;
            _callback();
        }
        finally
        {
```

```
        _isCallbackInProgress.Value = false;
    }
}
}
```

Если метод, обратный вызов которого осуществляет `Notify`, попытается сделать еще один вызов `Notify`, эта попытка рекурсии будет заблокирована с вызовом исключения. Тем не менее из-за использования `ThreadLocal<bool>` для отслеживания того, выполняется ли вызов, возможны одновременные вызовы, если каждый из них происходит в отдельном потоке.

Через свойство `Value` вы получаете и устанавливаете значение, которое `ThreadLocal<T>` содержит для текущего потока. Конструктор содержит перегрузку, и вы можете передать `Func<T>`, который будет вызываться каждый раз для создания начального значения по умолчанию, когда новый поток впервые попытается получить значение. (Инициализация здесь медлительная — обратный вызов не будет выполняться каждый раз, когда запускается новый поток. `ThreadLocal<T>` осуществляет обратный вызов только в первый раз, когда поток пытается использовать значение.) Фиксированного ограничения на количество объектов `ThreadLocal<T>`, которые вы можете создать, нет.

Также `ThreadLocal<T>` обеспечивает определенную поддержку межпотоковой связи. Если передать аргумент `true` одной из перегрузок конструктора, которая принимает `bool`, объект будет заставлять коллекцию сообщать о сохраненном для каждого потока последнем значении, доступном через ее свойство `Values`. Эта возможность предоставляется только в том случае, если вы запрашиваете ее при создании объекта, поскольку она требует определенной дополнительной работы. Кроме того, если вы используете в качестве аргумента типа ссылочный тип, то включение отслеживания может означать, что объекты будут сохраняться действительными дольше. Обычно любая ссылка, которую поток хранит в `ThreadLocal<T>`, прекратит существование в момент, когда поток завершится, и, если эта ссылка на конкретный объект была единственной, сборщик мусора сможет восстановить память. Если включить отслеживание, то все такие ссылки будут оставаться достижимыми до тех пор, пока достичим сам экземпляр `ThreadLocal<T>`, потому что `Values` возвращает значения даже для завершившихся потоков.

Есть кое-что, с чем следует проявлять осторожность при работе с локальным хранилищем потока. Если для каждого потока вы создаете новый объект,

имейте в виду, что приложение за свое время работы может создать большое количество потоков, особенно если вы используете пул потоков (который будет подробно описан ниже). Если создаваемые вами в каждом потоке объекты затратны, это может повлечь за собой проблемы. Кроме того, если в каждом потоке есть какие-либо определенные разовые ресурсы, вы не обязательно узнаете, когда такой поток завершится. Пул потоков регулярно создает и удаляет потоки, не сообщая вам, когда это происходит.

Последнее предостережение: будьте осторожны с локальным хранилищем потока (и любым основанным на нем механизмом), если планируете использовать функции асинхронного языка, описанные в главе 17, поскольку они позволяют в ходе работы в единственном вызове метода использовать несколько разных потоков. В таком методе использование внешних транзакций или чего-либо, что полагается на локальное состояние потока, будет плохой идеей. Многие функции .NET, которые, как можно подумать, могли бы использовать локальное хранилище потока (например, статическое свойство `HttpContext.Current` платформы ASP.NET Core, возвращающее объект, относящийся к HTTP-запросу, обрабатываемому текущим потоком), вместо этого связывают информацию с так называемым контекстом выполнения. Контекст выполнения является более гибким, поскольку при необходимости может перемещаться между потоками. Подробности чуть позже.

Чтобы проблемы, которые я только что обсуждал, вообще возникли, нам нужно иметь несколько потоков. Существует четыре основных способа использования многопоточности. При первом способе код выполняется в среде, которая создает несколько потоков от вашего имени, например ASP.NET Core. При другом используются определенные виды API на основе обратных вызовов. Несколько общих шаблонов для этого описаны в разделах «Задачи» на с. 857 и «Другие асинхронные шаблоны» на с. 873. Но два самых непосредственных способа использования потоков — это явное создание новых потоков или использование пула потоков .NET.

Класс `Thread`

Как я упоминал ранее, класс `Thread` (определенный в пространстве имен `System.Threading`) представляет собой поток CLR. С помощью свойства `Thread.CurrentThread` можно получить ссылку на объект `Thread`, представляющий поток, который выполняет ваш код, но если вам требуется многопоточность, вы можете просто создать новый объект `Thread`.

Новый поток должен знать, какой код он должен запускать при запуске, поэтому вы должны предоставить делегат, а поток при запуске вызовет метод, на который этот делегат ссылается. Поток будет работать до тех пор, пока этот метод не вернется обычным образом или пока исключение не будет передано на самый верх стека (или поток принудительно завершится через любой из механизмов ОС закрытия потоков или их содержащих процессов). В листинге 16.4 создаются три потока для одновременной загрузки содержимого трех веб-страниц.

Листинг 16.4. Создание потоков

```
class Program
{
    private static void Main(string[] args)
    {
        var t1 = new Thread(MyThreadEntryPoint);
        var t2 = new Thread(MyThreadEntryPoint);
        var t3 = new Thread(MyThreadEntryPoint);

        t1.Start("https://endjin.com/");
        t2.Start("https://oreilly.com/");
        t3.Start("https://dotnet.microsoft.com/");
    }

    private static void MyThreadEntryPoint(object arg)
    {
        string url = (string) arg;

        using (var w = new WebClient())
        {
            Console.WriteLine($"Downloading {url}");
            string page = w.DownloadString(url);
            Console.WriteLine($"Downloaded {url}, length {page.Length}");
        }
    }
}
```

Конструктор `Thread` имеет перегрузку и способен принимать два типа делегатов. Для делегата `ThreadStart` требуется метод, который не принимает аргументов и не возвращает значений, но в листинге 16.4 метод `MyThreadEntryPoint` принимает один аргумент `object`, который соответствует другому типу делегата, `ParameterizedThreadStart`. Это позволяет при вызове одного и того же метода в нескольких разных потоках передавать аргументы каждому потоку, что и показано в данном примере. Поток не будет

работать, пока вы не вызовете `Start`, и если вы используете тип делегата `ParameterizedThreadStart`, то должны вызвать перегрузку, которая принимает один аргумент объекта. Я использую это для того, чтобы каждый поток осуществлял загрузку из отдельного URL.



В большинстве случаев `HttpClient` предпочтительнее типа `WebClient`, показанного в листинге 16.4. Я сознательно избегаю использования `HttpClient`, потому что он предлагает только асинхронные методы, о которых мы поговорим позже.

Есть еще две перегрузки конструктора `Thread`, каждая из которых добавляет аргумент типа `int` после аргумента делегата. Этот `int` определяет размер стека для потока. Текущие реализации .NET требуют, чтобы в памяти стеки были непрерывными, что делает необходимым предварительное выделение адресного пространства. Если поток исчерпывает это пространство, CLR генерирует исключение `StackOverflowException`. (Вы обычно видите их только тогда, когда ошибка приводит к бесконечной рекурсии.) Без этого аргумента CLR будет использовать для процесса размер стека по умолчанию. (Размер зависит от ОС, и в Windows он обычно составляет 1 МБ. Вы можете изменить его, установив переменную среды `COMPlus_DefaultStackSize`.) Это случается не так уж часто, но все же случается. Если у вас имеется рекурсивный код, который создает очень глубокие стеки, для его запуска вам может потребоваться поток с большим стеком. И наоборот, если вы создаете огромное количество потоков, для экономии вы можете уменьшить размер стека, поскольку значение по умолчанию в 1 МБ обычно значительно больше, чем требуется на самом деле. Но следует понимать, что создание такого большого количества потоков обычно не очень хорошая идея. Таким образом, в большинстве случаев вы создаете умеренное количество потоков и вызываете конструкторы, которые используют размер стека по умолчанию.

Обратите внимание, что метод `Main` в листинге 16.4 завершается сразу после запуска трех потоков. Несмотря на это, приложение продолжает работать, и оно будет делать это до завершения всех потоков. CLR поддерживает процесс в активном состоянии до тех пор, пока запущены потоки переднего плана, тогда как поток переднего плана определен как любой поток, который явно не был обозначен как фоновый. Если вы хотите запретить конкретному потоку удерживать процесс, установите его свойство `IsBackground` в `true`. (Это

означает, что фоновые потоки могут быть прерваны во время выполнения чего-либо, поэтому будьте осторожны с тем, какую работу выполняете в них.)

Создание потоков напрямую – не единственный вариант. Часто используемой альтернативой является пул потоков.

Пул потоков

В большинстве операционных систем создание и закрытие потоков сравнительно затратно. Если нужно выполнить небольшую задачу (например, открыть веб-страницу или выполнить аналогичную краткую операцию), плохой идеей будет создавать поток только для этого и закрывать его после завершения работы. У такого подхода есть две серьезные проблемы: во-первых, вы можете потратить больше ресурсов на запуск и остановку, чем на полезную работу; во-вторых, если вы продолжаете создавать новые потоки по мере поступления дополнительной работы, система может замедлиться – при больших нагрузках создание еще большего числа потоков приведет лишь к снижению пропускной способности. Это связано с тем, что в дополнение к базовым накладным расходам на каждый поток (например, память, необходимая для стека) ОС должна регулярно переключаться между выполняемыми потоками, чтобы все они могли работать, а это переключение тоже имеет свою цену.



Пул потоков создает только фоновые потоки, поэтому, если пул потоков находится в процессе выполнения чего-либо на момент завершения последнего основного потока в вашем процессе, работа не будет завершена, поскольку в этот момент будут остановлены все фоновые потоки. Если нужно убедиться, что работа в пуле потоков завершена, необходимо подождать, пока это произойдет, и только потом завершить работу всех потоков переднего плана.

Чтобы можно было избежать этих проблем, .NET предоставляет пул потоков. У вас есть возможность предоставить делегат, который вызовет для потока из пула среда выполнения. При необходимости будет создан новый поток, но, если возможно, будет использоваться созданный ранее, что может поставить вашу задачу в очередь, если все созданные потоки пока заняты. После запуска вашего метода CLR обычно не завершает поток; вместо этого поток будет оставаться в пуле, ожидая, пока другие задачи захотят избежать стоимости создания потока для каждого из нескольких фрагментов работы.

При необходимости будут созданы новые потоки, но при этом пул попытается сохранить количество потоков на таком уровне, чтобы количество исполняемых потоков соответствовало количеству аппаратных потоков, так как это способно минимизировать затраты на переключение.

Запуск работы пула потоков с помощью Task

Обычный способ использования пула потоков — это класс `Task`. Это часть параллельной библиотеки задач (более подробно она обсуждается в разделе «Задачи» на с. 857), но ее базовое использование выглядит довольно просто, что показано в листинге 16.5.

Листинг 16.5. Выполнение кода в пуле потоков с помощью Task

```
Task.Run(() => MyThreadEntryPoint("https://oreilly.com/"));
```

Этот код ставит лямбду в очередь на выполнение в пуле потоков (при запуске будет просто вызван метод `MyThreadEntryPoint` из листинга 16.4). Если поток доступен, он начнет работу сразу же, но в ином случае задача будет ждать в очереди, пока поток не освободится (либо из-за завершения другого рабочего элемента в процессе, либо из-за добавления нового потока в пул потоков).

Есть и другие способы использования пула потоков, и наиболее очевидный из них — это класс `ThreadPool`. Его метод `QueueUserWorkItem` работает аналогично `StartNew` — вы передаете ему делегат и он помещает метод в очередь на выполнение. Это низкоуровневый API, который не предоставляет никакого прямого способа обработки завершения работы или связывания операций, поэтому в большинстве случаев предпочтительным является использование `Task`.

Эвристика создания потоков

.NET регулирует количество потоков в зависимости от рабочей нагрузки, которую вы даете. Используемая при этом эвристика не документирована и менялась в разных версиях .NET, поэтому не нужно полагаться на то конкретное поведение, которое я собираюсь описать. Тем не менее полезно хотя бы примерно знать, что происходит.

Если вы предоставляете пулу потоков только работу с привязкой к процессору, при которой каждый запрашиваемый для выполнения метод тратит все свое время на вычисления и никогда не блокируется в ожидании завершения ввода-вывода, то можете в итоге получить по одному потоку на

каждый аппаратный поток в вашей системе (хотя если отдельные рабочие элементы будут занимать достаточно много времени, пул потоков может решить выделить больше потоков). Например, на восьмиядерном двухпроцессорном гиперпоточном компьютере, на котором я пишу этот текст, постановка в очередь определенного объема рабочих элементов, интенсивно нагружающих ЦП, первоначально приводит к тому, что CLR создает 16 потоков пула потоков, и до тех пор, пока рабочие элементы завершаются примерно раз в секунду, количество потоков в основном остается на этом уровне. (Иногда число потоков возрастает из-за того, что среда выполнения время от времени пытается добавить дополнительный поток, чтобы выяснить, как это повлияет на пропускную способность, после чего показатель снова снижается.) Но если скорость, с которой программа переходит от элемента к элементу, падает, CLR постепенно увеличивает количество потоков.

Если потоки пула потоков блокируются (например, из-за ожидания данных с диска или ответа по сети от сервера), CLR будет наращивать количество потоков пула быстрее. Опять же, начинается все с одного потока на каждый аппаратный поток, но, когда медленные рабочие элементы используют очень мало процессорного времени, потоки могут добавляться не чаще двух раз в секунду.

В любом случае CLR в конечном итоге прекратит добавлять потоки. Точное ограничение по умолчанию для 32-разрядных процессов варьирует в зависимости от конкретной версии .NET, но обычно оно составляет порядка 1000 потоков. В 64-битном режиме значение по умолчанию составляет 32 767. Вы можете изменить это ограничение — класс `ThreadPool` содержит метод `SetMaxThreads`, который позволяет настраивать различные ограничения для вашего процесса. Вы можете столкнуться и с другими ограничениями, которые на практике устанавливают более низкий предел. Например, каждый поток имеет свой собственный стек, который должен занимать непрерывный диапазон виртуального адресного пространства. По умолчанию каждый поток получает 1 МБ адресного пространства процесса, зарезервированного для его стека, поэтому к тому времени, когда у вас будет 1000 потоков, вы будете использовать 1 ГБ адресного пространства для размещения одних лишь стеков.

32-разрядные процессы имеют только 4 ГБ адресного пространства, поэтому у вас может закончиться место для стеков запрашиваемых потоков².

² В 32-разрядных версиях Windows часть будет зарезервирована для системы, т. е. приложения могут использовать не более 3 ГБ диапазона адресов.

В любом случае 1000 потоков — это, как правило, более чем достаточное количество, поэтому если оно достигнуто, то это может служить сигналом некой фундаментальной проблемы, к которой вам следует обратиться. Таким образом, если вы вызываете `SetMaxThreads`, обычно следует указывать нижний предел — вы можете обнаружить, что при некоторых рабочих нагрузках ограничение количества потоков повышает пропускную способность за счет снижения уровня конкуренции за системные ресурсы.

`ThreadPool` также имеет метод `SetMinThreads`. Он позволяет гарантировать, что количество потоков не упадет ниже определенного числа. Это может пригодиться в приложениях, которые наиболее эффективно работают с некоторым минимальным количеством потоков и хотят иметь возможность сразу работать с максимальной скоростью, не дожидаясь, пока эвристика пула потоков отрегулирует число потоков.

Потоки завершения ввода-вывода

В Windows пул потоков содержит два вида потоков: рабочие потоки и потоки завершения ввода-вывода. Рабочие потоки используются для выполнения делегатов, которые вы ставите в очередь с помощью показанных мной техник запуска задач (но как я позже покажу в разделе «Планировщики» на с. 867, вы можете выбрать различные стратегии потоков). Класс `ThreadPool` также использует эти потоки посредством своего метода `QueueUserWorkItem`. Потоки завершения ввода-вывода используются в Windows для вызова методов, которые вы предоставляете в качестве обратных вызовов для момента, когда завершается инициированная асинхронно операция ввода-вывода (например, чтение данных из файла или сокета).

Внутри версия CLR для Windows использует механизм порта завершения ввода-вывода, который Windows предоставляет в целях эффективной обработки большого числа одновременных асинхронных операций. Пул потоков отделяет потоки, обслуживающие этот порт завершения, от других рабочих потоков. Это уменьшает шансы взаимоблокировки системы при достижении максимального ограничения на потоки в пуле. Если CLR не будет разделять потоки ввода-вывода, она может войти в состояние, когда все потоки пула потоков будут заняты, ожидая завершения ввода-вывода. В этот момент процесс окажется заблокированным, потому что не останется ни одного потока для обслуживания завершения операций ввода-вывода, ожидаемых другими потоками. (В Unix такого механизма нет, поэтому любой запрос на постановку в очередь в потоке ввода-вывода будет просто перенаправлен в пул рабочих потоков.)

На практике вы обычно можете смело игнорировать различие между потоками ввода-вывода и обычными потоками в пуле потоков, потому что CLR самостоятельно решает, какой из них использовать. Однако вы иногда будете сталкиваться с этим различием. Например, если вам по какой-то причине потребуется изменить размер пула потоков, вам придется отдельно указать верхние пределы для обычных потоков и потоков завершения ввода-вывода. Метод `SetMaxThreads`, о котором я упоминал в предыдущем разделе, принимает эти два аргумента.

Привязка к потоку и синхронизация контекста

Некоторые объекты требуют, чтобы их использовали только из определенных потоков. Особенно часто это встречается в коде пользовательского интерфейса — инфраструктуры пользовательского интерфейса WPF и Windows Forms требуют использования объектов пользовательского интерфейса из потока, в котором они были созданы. Это называется привязкой к потоку, и хотя чаще всего эта привязка связана с пользовательским интерфейсом, она также может возникать и в сценариях взаимодействия — некоторые COM-объекты тоже имеют привязку к потокам.

Привязка к потоку может осложнить жизнь, если требуется написать многопоточный код. Предположим, что вы тщательно проработали многопоточный алгоритм, который способен использовать все аппаратные потоки на компьютере конечного пользователя, значительно улучшая производительность при работе на многоядерном процессоре по сравнению с однопоточным алгоритмом. После завершения алгоритма может понадобиться представить результаты конечному пользователю. Привязка объектов пользовательского интерфейса к потоку требует, чтобы вы выполнили этот последний шаг в определенном потоке, но ваш многопоточный код вполне может выдать свои результаты в какой-то другой поток. (Скорее всего, вы изо всех сил старались избегать использования потока пользовательского интерфейса для работы, интенсивно нагружающей процессор, пытаясь добиться того, чтобы пользовательский интерфейс оставался максимально отзывчивым во время выполнения работы.) Если вы попытаетесь обновить пользовательский интерфейс из какого-то случайного рабочего потока, инфраструктура пользовательского интерфейса вызовет исключение, жалуясь на то, что вы нарушили его требования к привязке потоков. Так или иначе, вам потребуется передать сообщение обратно в поток пользовательского интерфейса, чтобы он мог отобразить результаты.

Для таких сценариев библиотека классов .NET предоставляет вспомогательный класс `SynchronizationContext`. Его статическое свойство `Current` возвращает экземпляр класса `SynchronizationContext`, представляющего контекст, в котором в данный момент выполняется ваш код. Например, в приложении WPF если вы запросите это свойство во время работы в потоке пользовательского интерфейса, то получите объект, связанный с этим потоком. Вы можете сохранить возвращаемый `Current` объект и использовать его из любого потока в любое время, когда вам потребуется выполнить следующую задачу с использованием потока пользовательского интерфейса. В листинге 16.6 это делается для того, чтобы он мог выполнять потенциально медленную работу в потоке из пула потоков, а затем обновлять пользовательский интерфейс в потоке пользовательского интерфейса.

Листинг 16.6. Использование пула потоков, а затем `SynchronizationContext`

```
private void findButton_Click(object sender, RoutedEventArgs e)
{
    SynchronizationContext uiContext = SynchronizationContext.Current;

    Task.Run(() =>
    {
        string pictures =
            Environment.GetFolderPath(
                Environment.SpecialFolder.MyPictures);
        var folder = new DirectoryInfo(pictures);
        FileInfo[] allFiles =
            folder.GetFiles("*.jpg", SearchOption.AllDirectories);
        FileInfo largest =
            allFiles.OrderByDescending(f => f.Length).FirstOrDefault();

        uiContext.Post(_ =>
        {
            long sizeMB = largest.Length / (1024 * 1024);
            outputTextBox.Text =
                $"Largest file ({sizeMB}MB) is {largest.FullName}";
        },
        null);
    });
}
```

Этот код обрабатывает событие `Click` кнопки. (Так получилось, что это приложение WPF, но `SynchronizationContext` работает точно так же и в других инфраструктурах пользовательского интерфейса, таких как Windows Forms.) Элементы пользовательского интерфейса вызывают собственные события

в потоке пользовательского интерфейса, поэтому когда первая строка обработчика щелчка мыши получает текущий `SynchronizationContext`, она получает контекст потока пользовательского интерфейса. После этого код выполняет определенную работу в потоке пула потоков через класс `Task`. Код просматривает каждое изображение в папке `MyPictures` пользователя и ищет самый большой файл, поэтому это может занять некоторое время. Выполнять медленную работу в потоке пользовательского интерфейса — это очень плохая идея. Элементы пользовательского интерфейса, принадлежащие этому потоку, не смогут быстро реагировать на ввод пользователя, пока поток пользовательского интерфейса занят чем-то другим. Поэтому отправить это в пул потоков — правильно.

Проблема с использованием в данном случае пула потоков заключается в том, что после завершения работы мы оказываемся не в том потоке, из которого можно обновить пользовательский интерфейс. Код обновляет свойство `Text` текстового поля, и мы получили бы исключение, если бы попытались проделать это из потока пула потоков. Таким образом, когда работа завершается, код использует объект `SynchronizationContext`, полученный ранее, и вызывает его метод `Post`. Этот метод принимает делегат и организует его вызов в потоке пользовательского интерфейса. (На самом деле он отправляет пользовательское сообщение в очередь сообщений Windows, и когда основной цикл обработки сообщений потока пользовательского интерфейса получает это сообщение, он вызывает делегат.)



Метод `Post` не ожидает завершения работы. Метод, который будет ожидать этого, называется `Send`, но я бы рекомендовал его не использовать. Блокировка рабочего потока, в то время как он ожидает выполнения потока пользовательского интерфейса, может оказаться рискованным делом, потому что, если поток пользовательского интерфейса в настоящее время заблокирован, ожидая чего-то от рабочего потока, все приложение будет заблокировано. `Post` позволяет избежать этой проблемы, давая рабочему потоку и потоку пользовательского интерфейса работать одновременно.

Листинг 16.6 извлекает `SynchronizationContext.Current` пока находится в потоке пользовательского интерфейса и только после этого начинает работу в пуле потоков. Это важно, потому что это статическое свойство является контекстно зависимым — оно возвращает контекст для потока пользовательского интерфейса, только когда вы в нем находитесь. (Фак-

тически в WPF каждое окно может иметь свой собственный поток пользовательского интерфейса, поэтому невозможно иметь API, возвращающий поток пользовательского интерфейса, — их может быть несколько.) Если вы читаете это свойство из потока пула потоков, возвращаемый объект контекста не будет переправлять работу в поток пользовательского интерфейса.

Механизм `SynchronizationContext` является расширяемым, поэтому вы можете по желанию наследовать из него свой собственный тип и вызывать его статический метод `SetSynchronizationContext`, чтобы сделать ваш контекст текущим контекстом для потока. Это может пригодиться в сценариях юнит-теста — механизм позволяет вам писать тесты для проверки правильности взаимодействия объектов с `SynchronizationContext` без необходимости создания реального пользовательского интерфейса.

ExecutionContext

У класса `SynchronizationContext` есть кузен `ExecutionContext`. Он выполняет аналогичную задачу, позволяя захватить текущий контекст, а затем через некоторое время использовать его для запуска делегата в том же контексте, но при этом отличается в двух аспектах. Во-первых, он захватывает не то же самое. Во-вторых, он использует другой подход для восстановления контекста. `SynchronizationContext` часто запускает вашу работу в каком-то конкретном потоке, тогда как `ExecutionContext` всегда будет использовать ваш поток. Он просто гарантирует, что вся собранная контекстная информация будет доступна в этом потоке. Разницу можно изложить так: `SynchronizationContext` выполняет работу в существующем контексте, тогда как `ExecutionContext` предоставляет вам контекстную информацию.

Текущий контекст можно получить, вызвав метод `ExecutionContext.Capture`. Контекст выполнения не захватывает локальное хранилище потока, но включает в себя любую информацию в текущем логическом контексте вызова. Вы можете получить доступ к нему через класс `CallContext`, который содержит методы `LogicalSetData` и `LogicalGetData`, предназначенные для сохранения и извлечения пар имя/значение, или же через обертку верхнего уровня, `AsyncLocal<T>`. Эта информация обычно связана с текущим потоком, но если вы запускаете код в захваченном контексте выполнения, он сделает информацию из логического контекста доступной, даже если этот код полностью выполняется в каком-то другом потоке.



Немного смущает, что реализация `ExecutionContext` в .NET Framework захватывает текущий `SynchronizationContext`, поэтому в некотором смысле `ExecutionContext` является расширенной версией `SynchronizationContext`. Однако `ExecutionContext` не использует захваченный `SynchronizationContext` при вызове вашего делегата. Он лишь дает гарантию, что если код, выполненный через `ExecutionContext`, читает свойство `SynchronizationContext.Current`, он получает свойство `SynchronizationContext`, которое было актуальным на момент захвата `ExecutionContext`. И это не обязательно будет `SynchronizationContext`, в котором в данный момент выполняется поток! Этот недостаток дизайна был исправлен в .NET Core.

.NET использует класс `ExecutionContext` всякий раз, когда длительная работа, которая начинается в одном потоке, впоследствии продолжается в другом (как это происходит с некоторыми из асинхронных шаблонов, описанных далее в этой главе). Вам может потребоваться использовать контекст выполнения аналогичным образом, если напишите код, принимающий обратный вызов, который он будет осуществлять позже, возможно, из какого-то другого потока. Для этого нужно вызвать `Capture` для получения текущего контекста, который позже вы сможете передать методу `Run` для вызова делегата. Листинг 16.7 показывает `ExecutionContext` в деле.

Листинг 16.7. Использование `ExecutionContext`

```
public class Defer
{
    private readonly Action _callback;
    private readonly ExecutionContext _context;

    public Defer(Action callback)
    {
        _callback = callback;
        _context = ExecutionContext.Capture();
    }

    public void Run()
    {
        ExecutionContext.Run(_context, (unusedStateArg) =>
            _callback(), null);
    }
}
```

В .NET Framework один захваченный `ExecutionContext` не может использоваться одновременно несколькими потоками. Иногда вам может потребоваться вызов нескольких различных методов в определенном контексте, а в многопоточной среде нельзя гарантировать, что предыдущий метод был завершен до вызова следующего. Для такого сценария `ExecutionContext` содержит метод `CreateCopy`, который создает копию контекста, позволяя вам совершать несколько одновременных вызовов в эквивалентных контекстах. В .NET Core `ExecutionContext` стал неизменяемым, что означает, что данное ограничение больше не применяется и `CreateCopy` просто возвращает ссылку `this`.

Синхронизация

Иногда вам может потребоваться многопоточный код, в котором несколько потоков имеют доступ к одному и тому же состоянию. К примеру, в главе 5 я предположил, что сервер может использовать `Dictionary< TKey, TValue >` как часть кэша, чтобы избежать дублирования работы при получении нескольких похожих запросов. Несмотря на то что в ряде сценариев подобное кэширование может обеспечить значительные преимущества в производительности, в многопоточной среде оно представляет собой проблему. (И если вы работаете над серверным кодом с высокими требованиями к производительности, вам, скорее всего, понадобится более одного потока для обработки запросов.) В разделе «Безопасность потоков» (Threadsafety) документации класса словаря сказано следующее:

`Dictionary< TKey, TValue >` может поддерживать несколько процедур чтения одновременно до тех пор, пока коллекция не изменяется. Тем не менее перечисление коллекции по своей сути не является потокобезопасной процедурой. В редких случаях, когда перечисление конкурирует с доступом для записи, коллекция должна быть заблокирована на все время перечисления. Чтобы доступ к коллекции был обеспечен нескольким потокам для чтения и записи, необходима реализация собственной синхронизации.

На лучшее нельзя было и надеяться — подавляющее большинство типов в библиотеке классов .NET попросту вообще не поддерживают многопоточное использование экземпляров. Большинство типов поддерживают многопоточное использование на уровне класса, но отдельные экземпляры

должны использоваться по одному потоку за раз. `Dictionary< TKey, TValue >` более великодушен: он явно поддерживает несколько одновременных процедур чтения, что очень хорошо для нашего сценария кэширования. Однако при изменении коллекции мы должны не только гарантировать, что мы не пытаемся изменить ее из нескольких потоков одновременно, но и не должны в это время выполнять какие-либо операции чтения.

Другие классы обобщенных коллекций предоставляют аналогичные гарантии (в отличие от большинства других классов в библиотеке). Например, `List< T >`, `Queue< T >`, `Stack< T >`, `SortedDictionary< TKey, TValue >`, `HashSet< T >` и `SortedSet< T >` — все поддерживают одновременное использование только для чтения. (Опять же, если вы изменяете какой-либо экземпляр этих коллекций, вы должны убедиться, что никакие другие потоки не изменяют и не читают из этого же экземпляра в то же время.) Конечно, вы всегда должны сверяться с документацией, прежде чем пробовать многопоточное использование каких-либо типов. Имейте в виду, что обобщенные типы интерфейсов коллекции не гарантируют потокобезопасности — хотя `List< T >` поддерживает параллельное чтение, так работают далеко не все реализации `IList< T >`³. (Например, представьте себе реализацию, которая является оберткой для чего-то потенциально медленного, например содержимого файла. Для такой оболочки может оказаться разумным кэшировать данные с целью ускорить операции чтения. Чтение элемента из такого списка может изменить его внутреннее состояние, поэтому оно может завершиться ошибкой при одновременном выполнении из нескольких потоков, если код не предпринял шагов по самозащите.)

Если вы можете устроить так, что вам никогда не придется изменять структуру данных во время ее использования из многопоточного кода, поддержки параллельного доступа, предлагаемой многими классами коллекций, может оказаться достаточно. Но если некоторым потокам понадобится изменить общее состояние, вам придется координировать доступ к этому состоянию. Для этого .NET предоставляет различные механизмы синхронизации, которые вы можете использовать, чтобы при необходимости ваши потоки по очереди обращались к общим объектам. В этом разделе я опишу наиболее используемые.

³ На момент написания этой статьи документация не содержала ничего о потокобезопасности доступных только для чтения `HashSet< T >` и `SortedSet< T >`. Тем не менее Microsoft заверила меня, что они тоже поддерживают одновременное чтение.

Мониторы и ключевое слово lock

Первым вариантом синхронизации многопоточного использования общего состояния является класс `Monitor`. Он популярен за счет своей эффективности и понятной модели, а C# обеспечивает прямую языковую поддержку, что делает его очень простым в использовании. В листинге 16.8 показан класс, который использует ключевое слово `lock` (которое, в свою очередь, использует класс `Monitor`) каждый раз, когда читает или изменяет свое внутреннее состояние. Это гарантирует, что только один поток будет иметь доступ к этому состоянию в данный момент.

Листинг 16.8. Защита состояния с помощью `lock`

```
public class SaleLog
{
    private readonly object _sync = new object();

    private decimal _total;

    private readonly List<string> _saleDetails = new List<string>();

    public decimal Total
    {
        get
        {
            lock (_sync)
            {
                return _total;
            }
        }
    }

    public void AddSale(string item, decimal price)
    {
        string details = $"{item} sold at {price}";
        lock (_sync)
        {
            _total += price;
            _saleDetails.Add(details);
        }
    }

    public string[] GetDetails(out decimal total)
    {
        lock (_sync)
```

```
{  
    total = _total;  
    return _saleDetails.ToArray();  
}  
}  
}
```

Чтобы использовать ключевое слово `lock`, вам необходимо предоставить ссылку на объект и блок кода. Компилятор C# генерирует код, который заставляет CLR гарантировать, что внутри блока `lock` для этого объекта одновременно находится не более одного потока. Предположим, вы создали один экземпляр этого класса `SaleLog` и в одном потоке вы вызвали метод `AddSale`, а в другом в то же время вызвали `GetDetails`. Оба потока достигнут операторов `lock`, передав одно и то же поле `_sync`. В зависимости от того, какой поток будет первым, ему будет разрешено запустить блок после `lock`. Другой поток будет вынужден ожидать — ему не будет позволено войти в собственный блок `lock`, пока первый поток не покинет свой.

Класс `SaleLog` использует любое из своих полей только изнутри блока блокировки с использованием аргумента `_sync`. Это гарантирует, что весь доступ к полям сериализуется (в смысле параллелизма — потоки получают доступ к полям по одному, но не ко всем одновременно). Когда метод `GetDetails` читает из полей `_total` и `_saleDetails`, он всегда получает согласованное представление — итоговое значение будет соответствовать текущему содержимому списка подробностей о продажах, поскольку код, который изменяет эти две части данных, делает это в пределах одного блока `lock`. Это означает, что обновления будут казаться атомарными с точки зрения любого другого блока `lock`, использующего `_sync`.

Использование блока `lock` даже для метода доступа `get`, который возвращает итоговое значение, может показаться излишним. Однако `decimal` — это 128-битное значение, поэтому доступ к данным этого типа не является атомарным по своей сути — без этой блокировки было бы возможно, чтобы возвращаемое значение состояло из смеси двух или более значений, которые `_total` имел в разное время. (Например, младшие 64 бита могут иметь более старое значение, нежели верхние 64 бита.) Это часто называют разорванным чтением. CLR гарантирует атомарное чтение и запись только для типов данных, размер которых не превышает 4 байта, а также для ссылок, даже для платформы, где их размер превышает 4 байта. (Это гарантируется только для естественным образом выровненных полей, но в C# поля всегда

будут выровнены, если только вы умышленно не убрали их выравнивание в целях взаимодействия.)

Важная тонкость работы листинга 16.8 заключается в том, что всякий раз, когда он возвращает информацию о своем внутреннем состоянии, то возвращает копию. Тип свойства `Total` — это `decimal`, который является значимым типом, а значения всегда возвращаются в виде копий. Но, когда речь идет о списке записей, метод `GetDetails` вызывает `ToArrayList`, который создает новый массив, содержащий копию текущего содержимого списка. Было бы ошибкой возвращать ссылку напрямую в `_saleDetails`, потому что это позволило бы коду вне класса `SalesLog` получить доступ и изменить коллекцию без использования `lock`. Нам необходимо обеспечить синхронизацию всего доступа к этой коллекции, но мы теряем эту возможность, если наш класс раздает ссылки на свое внутреннее состояние.



Если вы пишете код, выполняющий многопоточную работу, которая в итоге останавливается, вполне допустимо делиться ссылками на состояние после остановки работы. Но, если многопоточные изменения объекта продолжаются, нужно убедиться, что использование состояния этого объекта со всех сторон защищено.

Ключевое слово `lock` принимает любую ссылку на объект, поэтому вас могло удивить, что я специально создавал объект. Неужели нельзя было передать вместо этого `this`? Это бы сработало, но проблема в том, что ссылка `this` не является `private`. Это та же ссылка, по которой внешний код использует ваш объект. Использование публичной функции вашего объекта для синхронизации доступа к частному состоянию нецелесообразно; другой код может решить использовать ссылку на ваш объект в качестве аргумента для некоторых совершенно не связанных блоков `lock`. В нашем случае это, скорее всего, не вызовет проблемы, но в случае более сложного кода концептуально не связанные части с параллельным поведением могут оказаться связанными таким образом, что `this` может вызвать проблемы с производительностью или даже взаимоблокировки. Из этого следует, что лучше защищать код и использовать в качестве аргумента `lock` то, к чему имеет доступ только ваш код. Конечно, я мог бы использовать поле `_saleDetails`, поскольку оно относится к объекту, к которому имеет доступ только мой класс. Но даже если вы защищаете код, вы не должны предполагать, что это сделают другие разработчики, поэтому в целом безопаснее избегать использования экзем-

пляра класса, который написали не вы, в качестве аргумента `lock`, потому что никогда нельзя быть уверенным, что он не использует свою ссылку `this` для собственных целей блокировки.

В любом случае тот факт, что вы можете использовать любую ссылку на объект, выглядит довольно странно. В большинстве механизмов синхронизации .NET в качестве опорной точки для синхронизации используется экземпляр определенного типа. (Например, если вы хотите использовать семантику блокировки чтения/записи, необходимо использовать экземпляр класса `ReaderWriterLockSlim`, а не просто какой-либо старый объект.) Класс `Monitor` (тот, что используется `lock`) — это исключение, которое восходит к старому требованию об определенном уровне совместимости с Java (где имеется похожий примитив блокировки). Это не имеет отношения к современной разработке .NET, поэтому эта особенность в настоящее время является историческим нюансом. Использование отдельного объекта, единственная задача которого — служить в качестве аргумента `lock`, добавляет лишь минимальные затраты ресурсов (в первую очередь по сравнению с затратами на блокировку) и, как правило, облегчает понимание особенностей управления синхронизацией.



Использовать значимый тип в качестве аргумента `lock` не получится — C# не даст это сделать, и на то есть веская причина. Компилятор выполняет неявное преобразование аргумента блокировки в `object`, который в случае ссылочных типов не требует от CLR каких-либо действий во время выполнения. Но, когда вы преобразуете значимый тип в ссылку на тип `object`, необходимо выполнить упаковку. Эта упаковка станет аргументом `lock` и превратится в проблему, потому что вы будете получать новый блок каждый раз, когда конвертируете значение в ссылку на `object`. Таким образом, каждый раз при выполнении `lock` блок будет получать новый объект, т. е. на практике никакой синхронизации происходить не будет. Вот почему компилятор не даст даже попытаться.

Расширение ключевого слова `lock`

Каждый блок `lock` превращается в код, который выполняет три вещи: во-первых, он вызывает `Monitor.Enter`, передавая аргумент, предоставленный вами для `lock`. Затем он пытается запустить код в блоке. И наконец, как правило, он вызывает `Monitor.Exit` после завершения блока. Но все не так

просто, и виноваты в этом исключения. `Monitor.Exit` по-прежнему будет вызываться, если код, который вы поместили в блок, генерирует исключение, но он должен учитывать вероятность того, что исключение вызовет и сам `Monitor.Enter`. Это будет означать, что код более не владеет блокировкой и поэтому не должен вызывать `Monitor.Exit`. Листинг 16.9 показывает, во что компилятор превращает блок `lock` из метода `GetDetails` листинга 16.8.

Листинг 16.9. Расширение блока `lock`

```
bool lockWasTaken = false;
var temp = _sync;
try
{
    Monitor.Enter(temp, ref lockWasTaken);
    {
        total = _total;
        return _saleDetails.ToArray();
    }
}
finally
{
    if (lockWasTaken)
    {
        Monitor.Exit(temp);
    }
}
```

`Monitor.Enter` — это API, который определяет, владеет ли какой-либо другой поток блокировкой, и если да, то заставляет текущий поток ждать. Если он вообще завершается, то, как правило, успешно. (Может случиться взаимоблокировка, и в этом случае он никогда не вернется.) Существует небольшая вероятность сбоя, вызванного исключением, например из-за нехватки памяти. Это довольно необычная ситуация, но сгенерированный код тем не менее учитывает ее. Именно поэтому код для переменной `lockWasTaken` выглядит окольным путем. (На практике, кстати, компилятор сделает эту переменную скрытой без доступного имени. Я дал ей имя, чтобы сделать код более понятным.) Метод `Monitor.Enter` гарантирует, что получение блокировки будет атомарным и будет обновлять флаг, указывающий на то, была ли блокировка снята. Это гарантирует, что блок `finally` попытается вызвать `Exit` тогда и только тогда, когда блокировка получена.

`Monitor.Exit` сообщает CLR, что нам больше не нужен эксклюзивный доступ к ресурсам, к которым мы синхронизируем доступ, поэтому если какие-

либо другие потоки ожидают данный объект внутри `Monitor.Enter`, метод позволит одному из них продолжить работу. Компилятор помещает все это в блок `finally` для гарантии, что при выходе из блока путем выполнения его до конца, возврата из середины или при возникновении исключения блокировка будет снята.

Тот факт, что блокировка вызывает `Monitor.Exit` для исключения, имеет две стороны. С одной стороны, это уменьшает вероятность взаимоблокировки, обеспечивая удаление блокировок при сбое. С другой стороны, если исключение происходит, когда вы находитесь в процессе изменения какого-либо общего состояния, система может находиться в несогласованном состоянии. Снятие блокировок в таком случае позволит другим потокам получить доступ к этому состоянию, что способно вызвать дальнейшие проблемы. В некоторых ситуациях было бы лучше в случае исключения оставить блокировки нетронутыми — заблокированный процесс может нанести меньше ущерба, чем тот, который работает в поврежденном состоянии. Более надежной стратегией является написание кода, который гарантирует непротиворечивость при возникновении исключений либо путем отката любых изменений, которые были сделаны, если исключение предотвращает законченное обновление, либо путем организации изменения состояния автоматным способом (например, помещая новое состояние в совершенно новый объект и заменяя им предыдущий только после полной его инициализации). Но это лежит за пределами возможностей компилятора по автоматизации.

Ожидание и уведомление

Класс `Monitor` способен на большее, нежели просто гарантировать, что потоки принимают его по очереди. Он позволяет потокам ждать уведомления от какого-либо другого потока. Если поток получил монитор для определенного объекта, он может вызвать `Monitor.Wait`, передав туда этот объект. Это вызывает двойной эффект: освобождает монитор и блокирует поток. Он будет заблокирован, пока какой-нибудь другой поток не вызовет `Monitor.Pulse` или `PulseAll` для того же объекта; поток должен иметь монитор, чтобы у него была возможность вызывать любой из этих методов. (`Wait`, `Pulse` и `PulseAll` выдают исключение, если вызвать их, не обладая соответствующим монитором.)

Если поток вызывает `Pulse`, это позволяет одному потоку, ожидающему в `Wait`, проснуться. Вызов `PulseAll` позволяет запустить все потоки, обслуживающие монитор этого объекта. В любом случае `Monitor.Wait` повторно

запрашивает монитор перед возвратом, поэтому даже если вы вызываете `PulseAll`, потоки будут активироваться по одному — второй поток не сможет выйти из `Wait`, пока первый поток, который это сделал, не освободит монитор. Фактически никакие потоки не могут вернуться из `Wait` до тех пор, пока поток, вызвавший `Pulse` или `PulseAll`, не снимет блокировку.

Листинг 16.10 использует `Wait` и `Pulse` для предоставления обертки вокруг `Queue<T>`, которая заставляет поток, извлекающий элементы из очереди, ожидать, если очередь пуста. (Пример только для иллюстрации — если вам нужна такая очередь, вам не нужно писать свой собственный вариант. Используйте встроенный тип `BlockingCollection<T>` или типы в `System.Threading.Channels`.)

Листинг 16.10. `Wait` и `Pulse`

```
public class MessageQueue<T>
{
    private readonly object _sync = new object();

    private readonly Queue<T> _queue = new Queue<T>();

    public void Post(T message)
    {
        lock (_sync)
        {
            bool wasEmpty = _queue.Count == 0;
            _queue.Enqueue(message);
            if (wasEmpty)
            {
                Monitor.Pulse(_sync);
            }
        }
    }

    public T Get()
    {
        lock (_sync)
        {
            while (_queue.Count == 0)
            {
                Monitor.Wait(_sync);
            }
            return _queue.Dequeue();
        }
    }
}
```

В данном примере монитор используется двумя способами. Во-первых, это ключевое слово `lock`, которое гарантирует, что в один момент только один поток использует `Queue<T>`, где хранятся поставленные в очередь элементы. Кроме того, используется механизм ожидания и уведомления, что позволяет потоку, который использует элементы, эффективно блокироваться, когда очередь пуста, а любому потоку, который добавляет новые элементы в очередь, пробуждать заблокированный поток чтения.

Время ожидания

Ожидаете ли вы уведомления или же просто пытаетесь получить блокировку, можно указать время ожидания, в течение которого вы готовы ждать выполнения операции. Для получения блокировки следует использовать другой метод, `TryEnter`, но при ожидании уведомления вы просто используете другую перегрузку. (Такое не поддерживается компилятором, поэтому нельзя просто использовать ключевое слово `lock`.) В обоих случаях вы можете передать либо `int`, представляющий максимальное время ожидания, в миллисекундах, либо значение `TimeSpan`. Обе перегрузки возвращают значение `bool`, указывающее, успешно ли выполнена операция.

Такой подход используется для избегания взаимоблокировки процесса, но если ваш код не может получить блокировку в течение заданного времени, придется решать, что делать дальше. Если ваше приложение не может получить блокировку, в которой нуждается, то оно попросту не сможет выполнять ту работу, которую собиралось. Прекращение процесса может быть единственным разумным вариантом, потому что зависание, как правило, является признаком ошибки. Если это происходит, возможно, ваш процесс уже в скомпрометированном состоянии. Некоторые разработчики используют менее строгий подход в получении блокировок и могут рассматривать взаимоблокировку как нормальное явление. В этом случае может оказаться целесообразным прервать любую текущую операцию и либо повторить попытку позже, либо просто зарегистрировать сбой, отказавшись от этой конкретной операции и продолжив выполнение любого другого процесса. Но это может оказаться рискованным ходом.

SpinLock

`SpinLock` представляет логическую модель, аналогичную методам `Enter` и `Exit` класса `Monitor`. (Он не поддерживает ожидание и уведомление.) Это значимый тип, так что в некоторых случаях он может уменьшить количество

объектов, которые должны быть выделены для поддержки блокировки, — для `Monitor`, например, требуется объект в куче. Однако он еще и проще: в нем используется только одна стратегия обработки конфликтов, тогда как `Monitor` запускается с той же стратегией, что и `SpinLock`, а затем через некоторое время переключается на стратегию с более высокими начальными издержками, но зато более эффективную в случае длительного ожидания.

Когда вы вызываете любой из методов `Enter` (класса `Monitor` или `SpinLock`), то если блокировка доступна, она будет получена очень быстро — затраты обычно составляют несколько инструкций процессора. Если блокировка уже удерживается другим потоком, CLR остается в цикле, который опрашивает блокировку (т. е. буквует), ожидая ее доступности. Если блокировка удерживается лишь в течение очень короткого времени, это может стать крайне эффективной стратегией, поскольку позволяет избежать вовлечения ОС и оказывается чрезвычайно быстрой в случае, когда блокировка доступна. Даже в случае конкуренции в многоядерной или многопроцессорной среде ожидание в цикле может оказаться наиболее эффективной стратегией, потому что если блокировка удерживается только в течение очень короткого промежутка времени (например, на время сложения двух значений `decimal`), потоку не придется долго буквовать, прежде чем блокировка снова станет доступной.

Разница между `Monitor` и `SpinLock` заключается в том, что `Monitor` в конечном итоге прервет проверки в цикле, вернувшись к использованию планировщика ОС. Это будет эквивалентно выполнению многих тысяч (возможно, даже сотен тысяч) инструкций процессора, поэтому `Monitor` начинает с того же подхода, что и `SpinLock`. Однако если блокировка остается недоступной в течение длительного времени, ожидание в цикле становится неэффективным — всего 1 мс пробуксовки на современных процессорах будет включать в себя миллионы проверок, и в этом случае выполнение тысяч инструкций для эффективной приостановки потока выглядит гораздо привлекательнее. (Ожидание в цикле также представляет проблему в одноядерных системах, потому что продолжение работы зависит от потока, удерживающего блокировку)⁴.

⁴ На машинах с одним аппаратным потоком, когда `SpinLock` входит в свой цикл, он сообщает планировщику ОС, что готов уступить контроль над процессором, так что другие потоки (возможно, включая тот, который в данный момент удерживает блокировку) могут работать дальше. Иногда `SpinLock` делает так даже на многоядерных системах, чтобы избежать некоторых проблем, вызываемых излишне долгим пребыванием в цикле ожидания.

У `SpinLock` нет резервной стратегии. В отличие от `Monitor`, он будет буквовать либо до тех пор, пока не получит блокировку, либо до истечения времени ожидания (если вы его задавали). По этой причине в документации рекомендуется не использовать `SpinLock`, если вы делаете определенные вещи с удержанием блокировки. В их число входит выполнение чего-либо еще, что может вызвать блокировку (например, ожидание завершения ввода-вывода), или вызов другого кода, который может сделать то же самое. Также не рекомендуется вызывать метод через механизм, при котором нельзя быть уверенным, какой именно код будет выполняться (например, через интерфейс, виртуальный метод или делегат), или даже выделять память. Если вы удаленно делаете что-то нетривиальное, лучше положиться на `Monitor`. Однако доступ к `decimal` достаточно прост, чтобы его можно было защитить с помощью `SpinLock`, как показано в листинге 16.11.

Листинг 16.11. Защита доступа к `decimal` с помощью `SpinLock`

```
public class DecimalTotal
{
    private decimal _total;

    private SpinLock _lock;

    public decimal Total
    {
        get
        {
            bool acquiredLock = false;
            try
            {
                _lock.Enter(ref acquiredLock);
                return _total;
            }
            finally
            {
                if (acquiredLock)
                {
                    _lock.Exit();
                }
            }
        }
    }

    public void Add(decimal value)
```

```
{  
    bool acquiredLock = false;  
    try  
    {  
        _lock.Enter(ref acquiredLock);  
        _total += value;  
    }  
    finally  
    {  
        if (acquiredLock)  
        {  
            _lock.Exit();  
        }  
    }  
}  
}
```

Из-за отсутствия поддержки компилятора приходится писать значительно больше кода, чем в случае с использованием `lock`. Усилия могут вообще не окупиться. Поскольку `Monitor` начинает со спин-блокировки, он, скорее всего, будет иметь аналогичную производительность, так что единственным преимуществом здесь будет то, что мы избежим выделения дополнительного объекта кучи для блокировки (`SpinLock` — это структура и поэтому располагается внутри блока кучи объекта `DecimalTotal`). Вы должны использовать `SpinLock`, только если профилирование показывает, что при реальных рабочих нагрузках он работает лучше, чем `Monitor`.

Блокировки чтения/записи

Класс `ReaderWriterLockSlim` предоставляет другую модель блокировки, нежели `Monitor` и `SpinLock`. В случае `ReaderWriterLockSlim` при получении блокировки вы указываете, собираетесь ли вы читать или записывать. Блокировка позволяет нескольким потокам осуществлять чтение одновременно. Однако когда поток запрашивает блокировку для записи, она временно блокирует все дальнейшие потоки, которые пытаются читать данные, и ожидает, пока все потоки, которые уже приступили к чтению, снимут свои блокировки. Только после этого будет предоставлен доступ потоку, которому требуется запись. Как только записывающий поток снимает блокировку, все потоки, которые ожидали чтения, возвращаются к работе. Это позволяет получить эксклюзивный доступ на запись, но также

и означает, что когда запись не происходит, все читающие потоки могут работать параллельно.



Имеется также класс `ReaderWriterLock`. Использовать его не стоит, потому что у него есть проблемы с производительностью даже тогда, когда отсутствует конкуренция за блокировку. Кроме того, он также делает не самый оптимальный выбор, когда потоки чтения и записи ожидают получения блокировки. Уже давно есть более новый класс `ReaderWriterLockSlim` (еще со временем .NET 3.5), и именно его рекомендуется использовать во всех сценариях. Старый класс не убирают исключительно для обратной совместимости.

Это же можно сказать про множество классов коллекций, встроенных в .NET. Как я описал ранее, они зачастую поддерживают несколько параллельных потоков чтения, но требуют, чтобы модификация выполнялась исключительно одним потоком за раз и чтобы никакие читающие потоки не были активными во время внесения этих изменений. Тем не менее эта блокировка не обязана быть выбором по умолчанию, когда у вас есть и чтение, и запись.

Несмотря на улучшения производительности, которые «тонкая» (*slim*, часть имени класса) блокировка демонстрирует по сравнению с предшествующим классом, получение блокировки все еще занимает больше времени, чем вход в монитор. Если вам требуется удерживать блокировку лишь в течение очень короткого времени, может оказаться выгоднее использовать монитор. Теоретическое улучшение, обеспечиваемое большим параллелизмом, может оказаться перечеркнуто дополнительной работой, необходимой для получения блокировки. Даже если вы удерживаете блокировку в течение значительного времени, блокировки чтения/записи имеют преимущества только в том случае, если обновления происходят лишь время от времени. Если у вас более или менее постоянный набор потоков, желающих изменить данные, вы вряд ли добьетесь какого-либо улучшения производительности.

Как и во всех решениях, ориентированных на производительность, если вы рассматриваете возможность использования `ReaderWriterLockSlim` вместо более простой альтернативы в виде обычного монитора, то для обоих случаев следует измерить производительность под реалистичной рабочей нагрузкой, что позволит увидеть, какое влияние окажут изменения, если таковое вообще будет.

Объекты событий

Нативный API Windows, Win32, всегда содержал примитив синхронизации, называемый *событием*. С точки зрения .NET имя не самое удачное, поскольку термин там означает нечто совершенно другое, как уже обсуждалось в главе 9. В этом разделе, когда я ссылаюсь на событие и явно не квалифицирую его как событие .NET, я имею в виду примитив синхронизации.

Класс `ManualResetEvent` предоставляет механизм, в котором один поток может ожидать уведомления от другого потока. Это работает по-другому, чем методы `Wait` и `Pulse` класса `Monitor`. Во-первых, не нужно обладать монитором или другой блокировкой, чтобы иметь возможность ожидать или сигнализировать о событии. Во-вторых, методы `Pulse` класса `Monitor` делают что-либо, только если в `Monitor.Wait` для данного объекта заблокирован хотя бы один другой поток. Если никто ничего не ожидает, это будет выглядеть, как будто `Pulse` никогда не вызывался. Но `ManualResetEvent` запоминает свое состояние — после установки в свободное состояние он уже не вернется в несигнальное, если только вы не сбросите его вручную, вызвав `Reset` (отсюда и его название). Это делает его полезным для сценариев, в которых поток А не может продолжить работу, пока какой-то поток В не закончит свою работу, выполнение которой займет непредсказуемое количество времени. Возможно, потоку А придется ждать, но возможно, что поток В уже завершит работу к моменту проверки. В листинге 16.12 используется эта техника для выполнения ряда пересекающихся задач.

Листинг 16.12. Ожидание завершения работы с помощью `ManualResetEvent`

```
static void LogFailure(string message, string mailServer)
{
    var email = new SmtpClient(mailServer);

    using (var emailSent = new ManualResetEvent(false))
    {
        object sync = new object();
        bool tooLate = false; // Prevent call to Set after a timeout
        email.SendCompleted += (s, e) =>
        {
            lock(sync) { if (!tooLate) { emailSent.Set(); } }
        };
        email.SendAsync("logger@example.com", "sysadmin@example.com",
                      "Failure Report", "An error occurred: " + message, null);

        LogPersistently(message);

        if (!emailSent.WaitOne(TimeSpan.FromMinutes(1)))
    }
```

```
{  
    LogPersistently("Timeout sending email for error: " +  
                    message);  
}  
  
lock (sync)  
{  
    tooLate = true;  
}  
}  
}
```

Этот метод по электронной почте отправляет системному администратору отчет об ошибке, используя класс `SmtpClient` из пространства имен `System.Net.Mail`. Он также вызывает внутренний метод (здесь не показан), который называется `LogPersistently`, чтобы записать информацию о сбое с помощью локального механизма журналирования. Поскольку обе операции могут занять некоторое время, код отправляет сообщение асинхронно — метод `SendAsync` возвращается немедленно, а класс вызывает событие .NET после отправки сообщения электронной почты. Это позволяет коду продолжить выполнение `LogPersistently` во время отправки электронного письма.

Зарегистрировав сообщение (и прежде, чем вернуться), метод ожидает отправки электронного письма, и именно здесь на помощь приходит `ManualResetEvent`. Передав `false` конструктору, я установил событие в исходное несвободное состояние. Но в обработчике почтового события `SendCompleted` .NET я вызываю метод `Set` события синхронизации, который переводит его в свободное состояние. (В реальном коде я бы еще проверил аргумент обработчика событий .NET, чтобы узнать, не было ли ошибки, но здесь я эту проверку опускаю, потому что она не имеет отношения к тому, что я хочу проиллюстрировать.) Наконец, я вызываю `WaitOne`, который будет блокироваться, пока событие не перейдет в свободное состояние. `SmtpClient` может настолько быстро выполнить свою работу, что к тому времени, когда мой вызов `LogPersistently` завершится, электронное письмо уже уйдет. Но это нормально — в подобном случае `WaitOne` немедленно возвращается, потому что `ManualResetEvent` остается в состоянии сигнального, поскольку вы уже вызвали `Set`. Таким образом, не имеет значения, какая часть работы завершается первой — постоянное ведение журнала или отправка электронной почты, в любом случае `WaitOne` позволит продолжить поток после отправки электронной почты. (О происхождении странного имени этого метода можно прочитать во врезке «`WaitHandle`» на с. 844.)

WAITHANDLE

В реализациях Windows .NET `ManualResetEvent` является оберткой вокруг объекта события Win32. Существует несколько других классов синхронизации, которые также являются обертками для базовых примитивов синхронизации ОС: `AutoResetEvent`, `Mutex` и `Semaphore`. Все они происходят от общего базового класса, `WaitHandle`. (В реализациях, отличных от Windows .NET, когда эквивалентные примитивы ОС недоступны напрямую, библиотека классов просто реализует эквивалентное поведение.)

`WaitHandle` может находиться в одном из двух состояний: свободном (`signaled`) или занятом (`unsignaled`). Точное значение этого варьирует в зависимости от конкретного примитива. Событие `ManualReset` становится свободным, когда вы вызываете `Set` (и остается в таком состоянии до явного сброса). `Mutex` находится в сработавшем состоянии, только если в данный момент им не владеет ни один из потоков. Несмотря на различия в интерпретации, ожидание `WaitHandle` всегда будет блокироваться, если оно в свободном состоянии, и не будет блокироваться, если в занятом.

С объектами синхронизации Win32 вы можете подождать, либо пока один элемент не перейдет в свободное состояние, либо пока в свободное состояние не перейдет один из нескольких объектов (или вообще все). Класс `WaitHandle` определяет методы `WaitOne`, `WaitAny` и `WaitAll`, соответствующие этим трем видам ожидания. С примитивами, где успешное ожидание приводит к обладанию (монопольному в случае `Mutex` или частичному в случае `Semaphore`), может возникнуть проблема с попыткой ожидания нескольких объектов — если два потока пытаются получить одни и те же объекты, но делают это в разном порядке, попытки пересекаются и возникает взаимоблокировка. Но с этим может разобраться `WaitAll` — порядок указания элементов не имеет значения, потому что метод получает их атомарно — он не позволит ни одному из ожиданий успешно завершиться, пока все они не сделают это одновременно. (Конечно, если один поток выполняет второй вызов `WaitAll`, не освобождая сначала все объекты, полученные в ходе более раннего вызова, взаимоблокировка будет все еще возможна. `WaitAll` поможет, только если вы можете получить все необходимое за один шаг.)

`WaitAll` не работает в потоке, который использует режим STA COM, из-за ограничения в базовом API Windows, от которого он зависит. Как я описал в главе 14, если точка входа вашей программы помечена `[STAThread]`, она будет использовать этот режим, как и любой поток, в котором размещены элементы пользовательского интерфейса.

Вы также можете использовать `WaitHandle` совместно с пулем потоков. Класс `ThreadPool` имеет метод `RegisterWaitForSingleObject`, который принимает любой `WaitHandle` и выполняет предоставленный вами обратный вызов, когда дескриптор становится свободным. Как я расскажу позже, в случае ряда типов, производных от `WaitHandle` (например, `Mutex`), это не самая хорошая идея.

Также существует класс `AutoResetEvent`. Как только один поток вернулся из ожидания такого события, он автоматически возвращается в несвободное состояние. Таким образом, вызов `Set` для этого события пропустит не более одного потока. Если вы вызываете `Set` один раз, пока нет ожидающих потоков, событие останется установленным, поэтому, в отличие от `Monitor`.`Pulse`, уведомление не будет потеряно. Тем не менее событие не поддерживает счетчик вызовов `Set` — если вы осуществляете такой вызов дважды и если ни один поток не ожидает события, все равно будет пропущен только первый поток, а сброс осуществится немедленно.

Оба типа событий лишь косвенно происходят из `WaitHandle`, через базовый класс `EventWaitHandle`. Вы можете напрямую это использовать, что позволит вам указать ручной или автоматический сброс с помощью аргумента конструктора. Но что более интересно, так это то, что `EventWaitHandle` позволяет работать, пересекая границы процессов (но только в Windows). Базовым объектам событий `Win32` могут быть присвоены имена, и если вы знаете имя события, созданного другим процессом, то можете открыть его, передав имя при создании `EventWaitHandle`. (Если ни одного события с указанным именем еще не существует, ваш процесс будет первым, который его создает.) В Unix не существует эквивалента именованных событий, поэтому вы получите `PlatformNotSupportedException`, если попытаетесь создать его в подобных средах. Сами типы использовать можно, но без указания имени.

Существует также класс `ManualResetEventSlim`. Однако, в отличие от «нетонкого» (*nonslim*) чтения/записи, `ManualResetEvent` не был заменен его «тонким» преемником, потому что лишь старый тип поддерживает межпроцессное использование. Основное преимущество класса `ManualResetEventSlim` в том, что если ваш код должен ждать только очень короткое время, он будет более эффективным, потому что он некоторое время будет осуществлять проверки (как и `SpinLock`). Это избавляет от необходимости использовать относительно дорогие службы планировщика ОС. Тем не менее в конечном итоге он отступит и обратится к более затратному механизму. (Но даже в этом случае он будет немного более эффективным, поскольку более легковесен за счет того, что не требует поддержки межпроцессных операций.) «Тонкой» версии автоматического события не существует, поскольку события с автоматическим сбросом — это не то, что можно часто увидеть.

Класс `Barrier`

В предыдущем разделе я показал, как можно использовать событие для организации параллельной работы, позволяя одному потоку ждать, пока не произойдет что-то еще, после чего продолжать работу. Библиотека классов содержит класс, который способен обрабатывать похожие типы координации, но с несколько другой семантикой. Класс `Barrier` может работать с несколькими участниками, а также поддерживает многофазность, что означает, что в ходе работы потоки могут ожидать друг друга по несколько раз. Класс `Barrier` является симметричным — тогда как в листинге 16.12 обработчик событий вызывает `Set`, в то время как другой поток вызывает `WaitOne`, в случае `Barrier` все участники вызывают метод `SignalAndWait`, который эффективно объединяет установку и ожидание в одну операцию.

Когда участник вызывает `SignalAndWait`, метод блокируется до тех пор, пока его не вызовут все участники, после чего все они будут разблокированы и смогут продолжить. `Barrier` знает, сколько участников ему ожидать, потому что вы передаете их число в качестве аргумента конструктора.

Многофазная операция просто включает в себя повторные обходы. Как только последний участник вызывает `SignalAndWait`, освобождая остальных, и если какой-либо поток повторно вызовет `SignalAndWait`, он будет блокироваться, как и раньше, пока все остальные тоже не вызовут его повторно. `CurrentPhaseNumber` сообщает вам, сколько раз это уже происходило.

Симметрия делает `Barrier` менее подходящим решением для листинга 16.12, чем `ManualResetEvent`, потому что в этом случае только один из потоков действительно должен ждать. Нет смысла заставлять обработчик событий `SendComplete` ожидать завершения постоянного обновления журнала — только одного из участников на самом деле заботит завершение работы. `ManualResetEvent` поддерживает только одного участника, но это не обязательно может быть причиной использования `Barrier`. Если вам нужна асимметрия в стиле событий с несколькими участниками, то есть еще один подход: обратные отсчеты.

`CountdownEvent`

Класс `CountdownEvent` похож на событие, но при этом позволяет указать, что он должен получить свободное состояние определенное количество

раз, прежде чем пропускать ожидающие потоки. Конструктор принимает начальный аргумент `count`, который вы можете увеличить в любое время, вызвав `AddCount`. Метод `Signal` вызывается, чтобы уменьшить счетчик; по умолчанию он уменьшает его на единицу, но есть перегрузка, которая позволяет вам уменьшить его на указанное число.

Метод `Wait` блокируется до тех пор, пока счетчик не достигнет нуля. Если вы хотите проверить текущий счетчик, то можно прочитать свойство `CurrentCount`.

Семафоры

Есть еще одна основанная на подсчете система под названием *семафор*, которая широко используется в параллельных системах. Windows имеет ее встроенную поддержку, и класс .NET `Semaphore` изначально разрабатывался как обертка для этой поддержки. Как и обертки событий, `Semaphore` проходит от `WaitHandle`, а на платформах, отличных от Windows, поведение эмулируется. В то время как `CountdownEvent` пропускает ожидающие потоки только после обнуления счетчика, семафор при обнулении счетчика начинает блокировать потоки. Его можно использовать, когда нужно убедиться, что определенную работу выполняет определенное количество потоков одновременно.

Поскольку `Semaphore` наследуется от `WaitHandle`, для ожидания вы вызываете метод `WaitOne`. Это приведет к блокировке, только если счетчик уже равен нулю. При завершении метод уменьшает счетчик на единицу. Вызывая `Release`, вы увеличиваете счетчик. Необходимо не только задать начальное значение в качестве аргумента конструктора, но и указать максимальное значение счетчика — если вызов `Release` пытается установить количество выше максимального, он выдаст исключение.

Как и в случае событий, Windows поддерживает кросс-процессное использование семафоров, поэтому вы можете при желании передать имя семафора в качестве аргумента конструктора. Это откроет существующий семафор или создаст новый, если семафор с указанным именем еще не существует.

Имеется также и класс `SemaphoreSlim`. Как и `ManualResetEventSlim`, он дает преимущество в производительности в случаях, когда потокам не приходится блокироваться надолго. `SemaphoreSlim` содержит два способа уменьшить счетчик. Его метод `Wait` работает так же, как и метод `WaitOne`

класса `Semaphore`, но, кроме этого, есть и `WaitAsync`, который возвращает `Task`, завершающуюся, когда счетчик не равен нулю (и уменьшает счетчик при завершении задачи). Это означает, что вам не нужно блокировать поток, пока вы ожидаете доступности семафора. Более того, это означает, что вы можете использовать ключевое слово `await`, описанное в главе 17, чтобы уменьшить значение семафора.

Мьютекс

Windows определяет примитив синхронизации под названием *мьютекс*, для которого .NET предоставляет класс-оболочку `Mutex`. Название является сокращением от «взаимоисключающий», потому что только один поток одновременно может обладать мьютексом — например, если им владеет поток А, потоку В это уже не позволено, и наоборот. Конечно, это именно то, что делает ключевое слово `lock` посредством класса `Monitor`, но `Mutex` имеет перед ним два преимущества. Он предлагает межпроцессную поддержку: как и в случае других примитивов межпроцессной синхронизации, вы можете передать имя при создании мьютекса. (И в отличие от всех остальных, данный тип поддерживает именование даже на платформах на основе Unix.) Кроме того, используя `Mutex`, вы можете ожидать несколько объектов в одной операции.



Метод `ThreadPool.RegisterWaitForSingleObject` не работает в случае мьютекса, поскольку Win32 требует, чтобы владение мьютексом было привязано к определенному потоку, а внутренняя работа пула потоков означает, что `RegisterWaitForSingleObject` не может определить, какой поток пула потоков обрабатывает обратный вызов с мьютексом.

Получить мьютекс можно, вызвав `WaitOne`, и, если в это время мьютексом владеет какой-то другой поток, `WaitOne` заблокируется, пока этот поток не вызовет `ReleaseMutex`. Как только `WaitOne` успешно завершится, мьютекс ваш. Вы должны освободить мьютекс из того же потока, в котором вы его получили.

«Тонкой» версии класса `Mutex` не существует. У нас уже есть эквивалент с минимальной стоимостью, потому что все объекты .NET обладают врожденной способностью обеспечивать дешевое взаимное исключение благодаря классу `Monitor` и ключевому слову `lock`.

Interlocked

Класс `Interlocked` немного отличается от других типов, которые я описал в этом разделе. Он поддерживает одновременный доступ к общим данным, но это не примитив синхронизации. Вместо этого он определяет статические методы, которые делают возможными атомарные формы различных простых операций.

Например, он предоставляет методы `Increment`, `Decrement` и `Add` с перегрузками, поддерживающими значения `int` и `long`. (Все они похожи, так как увеличение и уменьшение — это просто прибавление 1 или -1 .) Увеличение включает в себя считывание значения из некоторого места хранения, вычисление измененного значения и сохранение его обратно в то же место. Если вы используете для этого обычные операторы C#, то работа может нарушиться, если несколько потоков попытаются изменить одно и то же местоположение одновременно. Если значение изначально равно 0 и какой-то поток читает это значение, а затем другой поток, в свою очередь, делает то же самое, т. е. оба после этого добавляют 1 и сохраняют результат, то каждый из них в конечном итоге запишет снова 1 — два потока, попытавшись увеличить значение, увеличат его только на единицу. `Interlocked`-форма этих операций предотвращает подобное перекрытие.

`Interlocked` также предлагает различные методы обмена значениями. Метод `Exchange` принимает два аргумента: ссылку на значение и значение. Он возвращает значение, которое в настоящее время находится в местоположении, на которое ссылается первый аргумент, а также перезаписывает это местоположение значением, предоставленным в качестве второго аргумента, причем выполняет эти два шага как одну атомарную операцию. У метода имеются перегрузки, поддерживающие `int`, `long`, `object`, `float`, `double` и тип с именем `IntPtr`, который представляет собой неуправляемый указатель. Существует также обобщенный тип `Exchange<T>`, где `T` может быть любым ссылочным типом.

Имеется и поддержка условного обмена с помощью метода `CompareExchange`. Он принимает три значения — как и в случае `Exchange`, он ссылается на некоторую переменную, которую вы хотите изменить, и на значение, которым вы хотите ее заменить, но также принимает и третий аргумент: значение, которое, по вашему мнению, уже там находится. Если значение в месте хранения не соответствует ожидаемому, метод не будет ничего менять. (Он по-прежнему возвращает любое значение, которое было в этом месте хра-

нения, изменил он его или нет.) На самом деле в терминах этой операции возможно реализовать и другие атомарные операции, которые я описал. В листинге 16.13 реализуется атомарная операция увеличения.

Схема будет повторяться и для других операций: прочитать текущее значение, вычислить значение, на которое его следует заменить, после чего заменить, но только в случае, если значение за это время не изменилось. Если значение изменяется между извлечением текущего значения и его заменой, придется начинать заново. Здесь требуется осторожность — даже если `CompareExchange` завершится успешно, возможно, что другие потоки изменили значение дважды между вашим чтением значения и его обновлением, а второе обновление вернуло все в состояние, которое было до первого. В случае сложения и вычитания это не имеет большого значения, потому что не влияет на результат, но в целом не стоит слишком полагаться на успешное обновление. Если имеются сомнения, часто лучше придерживаться какого-то из более тяжеловесных механизмов синхронизации.

Листинг 16.13. Использование CompareExchange

```
static int InterlockedIncrement(ref int target)
{
    int current, newValue;
    do
    {
        current = target;
        newValue = current + 1;
    }
    while (Interlocked.CompareExchange(ref target, newValue, current)
           != current);
    return newValue;
}
```

Простейшей операцией в `Interlocked` является метод `Read`. Он принимает `ref long` и считывает значение атомарно относительно любых других операций над 64-битными значениями, которые вы выполняете посредством `Interlocked`. Это позволяет безопасно читать 64-битные значения — в общем случае CLR не гарантирует, что операции чтения 64-битных значений будут атомарными. (В 64-битном процессе они обычно таковыми и будут, но если вы хотите атомарности в 32-битных архитектурах, вам необходимо использовать `Interlocked.Read`.) Для 32-битных значений перегрузок нет, потому что их чтение и запись всегда атомарны.

Операции, поддерживаемые `Interlocked`, соответствуют элементарным операциям, которые большинство процессоров способны поддерживать более или менее напрямую. (Некоторые архитектуры ЦП на низком уровне не поддерживают все операции, в то время как другие только сравнение и обмен, строя все остальные операции на основе этих. Но в любом случае эти операции представляют собой не более нескольких инструкций.) Это означает, что они достаточно эффективны. Они значительно затратнее, чем выполнение эквивалентных операций без блокировки в обычном коде, потому что атомарные инструкции ЦП должны координироваться по всем ядрам ЦП (и по всем чипам ЦП компьютеров, на которых установлено несколько физически отдельных ЦП), чтобы гарантировать эту атомарность. Тем не менее они берут на себя часть затрат, когда оператор `lock` в конечном итоге блокирует поток на уровне ОС.

Эти виды операций иногда описываются как *свободные от блокировки*. Это не совсем точный термин — компьютер все же запрашивает блокировки, хотя и очень быстро и на довольно низком уровне аппаратного обеспечения. Атомарные операции чтения-изменения-записи фактически получают монополию на использование памяти компьютера для двух циклов шины. Однако блокировок ОС не случается, планировщику не нужно вмешиваться, а сами блокировки удерживаются в течение очень короткого времени — часто только для одной инструкции машинного кода. Что более важно, используемая здесь узкоспециализированная и низкоуровневая форма блокировки не позволяет удерживать одну блокировку в ожидании получения другой — за раз код может блокировать только что-то одно. Это означает, что такого рода операции не будут приводить ко взаимоблокировке. Тем не менее простота, которая исключает взаимные блокировки, имеет свои плюсы и минусы.

Недостатком взаимосвязанных операций является то, что атомарность применяется только к чрезвычайно простым операциям. Очень сложно построить более сложную логику так, чтобы она работала правильно в многопоточной среде, и при этом использовать только `Interlocked`. Использовать высокоуровневые примитивы синхронизации проще и значительно менее рискованно, поскольку они позволяют довольно просто защищать более сложные операции, а не только отдельные вычисления. Обычно `Interlocked` следует использовать только в чрезвычайно чувствительной к производительности работе, и даже в этом случае вам следует тщательно все замерить, чтобы убедиться в ожидаемом эффекте, — такой

код, как показан в листинге 16.13, до завершения теоретически может циклически повторяться любое количество раз, так что затраты могут оказаться больше ожидаемых.

Одна из самых больших неприятностей, связанных с написанием правильного кода при использовании низкоуровневых атомарных операций, — это то, что вы можете столкнуться с проблемами, вызванными работой кэшей ЦП. Работа, выполняемая одним потоком, может не сразу стать видимой другим потокам, а в ряде случаев доступ к памяти не обязательно будет происходить в порядке, заданном вашим кодом. Использование высокоуровневых примитивов синхронизации позволяет обойти эти проблемы, применяя определенные ограничения сортировки. Но если вместо этого для создания собственных механизмов синхронизации вы решите использовать `Interlocked`, вам нужно будет понимать определяемую .NET модель памяти, в которой несколько потоков обращаются к одной и той же памяти одновременно, а вам нужно использовать либо метод `MemoryBarrier`, определенный классом `Interlocked`, либо методы, определенные классом `Volatile`, служащие для обеспечения безошибочной работы. Это выходит за рамки данной книги, но является прекрасным способом написания кода, который выглядит так, как будто он работает, но ломается при большой нагрузке (т. е. когда это, скорее всего, имеет наибольшее значение). Поэтому такие методы редко стоят затрат на их реализацию. Придерживайтесь других механизмов, которые я обсуждал в этой главе, если только вы не оказались в безвыходной ситуации.

Отложенная инициализация

Когда требуется, чтобы объект был доступен из нескольких потоков, и если этот объект может быть неизменным (т. е. его поля не меняются после создания), часто можно избежать необходимости в синхронизации. В случае нескольких потоков считывать данные из одного и того же места одновременно всегда безопасно. Проблема возникает только в случае, если необходимо изменить данные. Однако есть одна проблема: когда и как нужно инициализировать общий объект? Одним из решений может быть сохранение ссылки на объект в статическом поле, инициализированном из статического конструктора или инициализатора поля, — CLR гарантирует единственный запуск статической инициализации для любого класса. Однако это может привести к тому, что объект будет создан раньше, чем нужно. Если вы вы-

полняете слишком много работы при статической инициализации, это может отрицательно сказаться на продолжительности запуска приложения.

Возможно, вы захотите повременить с инициализацией до тех пор, пока объект впервые не понадобится. Это и называется *отложенной, или ленивой, инициализацией*. Реализовать ее несложно — можно просто проверить поле, чтобы понять, является ли оно пустым, и инициализировать его, если это не так. При этом можно использовать блокировку, чтобы гарантировать, что только один поток получит право создать значение. Тем не менее это та область, в которой разработчики, похоже, больше всего склонны показывать всем, какие они умные, что потенциально может привести к нежелательной демонстрации того, что они не так умны, как думают. Ключевое слово `lock` работает довольно эффективно, но с помощью `Interlocked` можно добиться гораздо большего. Однако детали переупорядочения доступа к памяти в многопроцессорных системах делают легким написание кода, который выполняется быстро, выглядит умно и не всегда работает. Для борьбы с этой неизбывной проблемой .NET предоставляет два класса для выполнения отложенной инициализации без использования блокировки или других потенциально дорогих примитивов синхронизации. Самый простой в использовании — `Lazy<T>`.

Lazy<T>

Класс `Lazy<T>` предоставляет свойство `Value` типа `T`, и он не будет создавать возвращаемый `Value` экземпляр, пока свойство не будет прочитано в первый раз. По умолчанию `Lazy<T>` будет использовать для `T` конструктор без аргументов, но вы можете предоставить аргумент обратного вызова, который позволит вам задать свой собственный метод создания экземпляра.

`Lazy<T>` может помочь вам справиться с условиями гонки. На самом деле вы можете задать тот уровень многопоточной защиты, который вам требуется. Поскольку отложенная инициализация также может быть полезна и в однопоточных средах, можно полностью отключить поддержку многопоточности (передав `false` или `LazyThreadSafetyMode.None` в качестве аргумента конструктора). Но для многопоточных сред вы можете выбрать один из двух других режимов в перечислении `LazyThreadSafetyMode`. Они определяют, что произойдет, если все потоки попытаются впервые прочитать свойство `Value` более или менее одновременно. `PublicationOnly` не пытается гарантировать, что только один поток создает объект, — он применяет какую-либо синхронизацию лишь в тот момент, когда поток

завершает создание объекта. Первый поток, завершивший построение или инициализацию, получает объект, а те, что были созданы любыми другими потоками, начавшими инициализацию, отбрасываются. Как только значение станет доступным, все дальнейшие попытки прочитать `Value` будут просто его возвращать. Если вы выберете `ExecutionAndPublication`, только один поток будет иметь возможность попытаться создать его. Это может показаться менее расточительным, но у `PublicationOnly` есть потенциальное преимущество: поскольку он избегает удержания каких-либо блокировок во время инициализации, вы с меньшей вероятностью допустите ошибки, приводящие к взаимоблокировке, в то время как сам код инициализации пытается получить какие-либо блокировки. Кроме того, `PublicationOnly` по-другому обрабатывает ошибки. Если при первой попытке инициализации возникает исключение, другим потокам, начавшим попытку построения, дается возможность завершиться. Но в случае `ExecutionAndPublication`, если единственная попытка инициализации завершается неудачей, исключение сохраняется и будет выбрасываться каждый раз, когда код пытается читать значение.

LazyInitializer

Другим классом, поддерживающим отложенную инициализацию, является `LazyInitializer`. Это статический класс, и его необходимо использовать только через его статические обобщенные методы. Он немного сложнее в использовании, чем `Lazy<T>`, но зато избавляет от необходимости выделять еще один объект в дополнение к экземпляру с отложенным выделением, который вам требуется. В листинге 16.14 показано, как его использовать.

Листинг 16.14. Использование LazyInitializer

```
public class Cache<T>
{
    private static Dictionary<string, T> _d;

    public static IDictionary<string, T> Dictionary =>
        LazyInitializer.EnsureInitialized(ref _d);
}
```

Если поле имеет значение `null`, метод `EnsureInitialized` создает экземпляр типа аргумента, в данном случае `Dictionary<string, T>`. В противном случае он вернет значение, которое уже находится в поле. Есть и другие перегрузки. Вы можете передать обратный вызов, так же как в случае `Lazy<T>`. Вы также можете передать аргумент `ref bool`, который будет проверен для выяснения,

происходила ли уже инициализация (и он устанавливает значение `true`, когда выполняет инициализацию).

Инициализатор статического поля дал бы нам такую же инициализацию, которая выполняется лишь однажды, но в результате запустился намного раньше относительно времени жизни процесса. В более сложном классе с несколькими полями статическая инициализация может даже привести к ненужной работе, потому что она касается всего класса, и в итоге вы будете создавать объекты, которые никогда не используются. Это может увеличить время, необходимое для запуска приложения. `LazyInitializer` позволяет инициализировать отдельные поля тогда, когда они впервые используются, при этом вы гарантированно выполняете только ту работу, которая необходима.

Другие виды поддержки параллелизма в библиотеке классов

Пространство имен `System.Collections.Concurrent` определяет различные коллекции, которые обладают большей надежностью в плане многопоточности, чем обычные коллекции, что означает, что вы можете использовать их без использования других примитивов синхронизации. Но будьте внимательны — как всегда, даже если отдельные операции имеют четко определенное поведение в многопоточном окружении, это не обязательно вам поможет, если операция, которую необходимо выполнить, состоит из нескольких шагов. Чтобы гарантировать согласованность, все еще может потребоваться координация в более широком контексте. Но в некоторых ситуациях параллельные коллекции могут обеспечить вас всем, что нужно.

В отличие от непараллельных коллекций, `ConcurrentDictionary`, `ConcurrentBag`, `ConcurrentStack` и `ConcurrentQueue` поддерживают модификацию своего содержимого, даже когда выполняется перечисление их содержимого (например, с помощью цикла `foreach`). Словарь предоставляет перечислитель в реальном времени в том смысле, что, если значения добавляются или удаляются, когда вы находитесь в процессе перечисления, он способен показать вам некоторые добавленные элементы и не отображать уже удаленные элементы. Но твердых гарантий нет, и не в последнюю очередь потому, что с многопоточным кодом, когда дело происходит в двух разных потоках, не всегда до конца ясно, что же произошло первым — законы относительности подсказывают, что это может зависеть от вашей точки зрения. Это означает, что перечислитель может вернуть элемент после удаления его

из словаря. Мультимножество, стек и очередь используют другой подход: все их перечислители делают снимок и перебирают уже его, поэтому цикл `foreach` будет видеть набор содержимого, который соответствует тому, что было в коллекции в какой-то момент в прошлом, даже если содержимое с тех пор изменилось.

Как я уже упоминал в главе 5, параллельные коллекции представляют API, которые похожи на их непараллельные аналоги, но с некоторыми дополнительными членами для поддержки атомарного добавления и удаления элементов.

Еще одна часть библиотеки классов, которая может помочь вам в работе с параллелизмом, при этом не требуя явного использования примитивов синхронизации, — это библиотека Rx (тема главы 11). Он предлагает различные операторы, которые способны объединять несколько асинхронных потоков в один поток. С помощью этого можно решить вопросы параллелизма, помня, что любой наблюдаемый объект будет предоставлять наблюдателям элементы по одному за раз. Rx делает все возможное для обеспечения того, чтобы оставаться в рамках этих правил, даже если он объединяет входные данные из множества отдельных потоков, поставляющих элементы одновременно. Он никогда не потребует от наблюдателя иметь дело с более чем одним элементом одновременно.

NuGet пакет `System.Threading.Channels` содержит типы, поддерживающие шаблоны «провайдер/потребитель», в которых один или несколько потоков генерируют данные, а другие — потребляют. Вы можете выбрать, буферизовать ли каналы, что позволит провайдерам опережать потребителей на определенное количество шагов. (Коллекция `BlockingCollection<T>` в `System.Collections.Concurrent` тоже делает что-то подобное. Однако она менее гибкая и не поддерживает ключевое слово `await`, описанное в главе 17.)

Наконец, в многопоточных сценариях стоит подумать об использовании классов неизменяемых коллекций, которые я описал в главе 5. Они поддерживают одновременный доступ из любого числа потоков, и, поскольку они неизменны, вопроса обработки одновременных запросов на запись просто не возникает. Очевидно, что неизменность накладывает заметные ограничения, но если вы отыщете способ работы с этими типами (напомню, что встроенный тип `string` является неизменяемым, так что у вас уже есть кое-какой опыт работы с неизменяемыми данными), они могут оказаться очень полезными в ряде параллельных сценариев.

Задачи

В этой главе я уже показал, как использовать класс `Task` для запуска работы в пуле потоков. Этот класс представляет собой нечто большее, нежели простую оболочку для пула потоков. `Task` и родственные ему типы образуют библиотеку параллельных задач (TPL) и могут обрабатывать более широкий спектр сценариев. Задачи особенно важны, потому что функции асинхронного языка C# (которые обсуждаются в главе 17) способны работать непосредственно с объектами задач. Многие API в библиотеке классов .NET предлагают асинхронную работу на основе задач.

Хотя задачи являются предпочтительным способом использования пула потоков, они применяются не только в многопоточности. Базовые абстракции гораздо более гибкие.

Классы `Task` и `Task<T>`

В основе TPL лежат два класса: `Task` и производный от него `Task<T>`. Базовый класс `Task` представляет собой некоторую задачу, выполнение которой может занять определенное время. `Task<T>` расширяет эту концепцию, представляя собой задачу, которая по завершении производит результат (типа `T`). (Необобщенный `Task` не дает никакого результата. Это асинхронный эквивалент типа возврата `void`.) Обратите внимание, что эти концепции не подразумевают обязательное использование потоков.

Большинство операций ввода-вывода могут занять некоторое время, и в большинстве случаев библиотека классов .NET предоставляет для них API на основе задач. В листинге 16.15 используется асинхронный метод для извлечения содержимого веб-страницы в виде строки. Поскольку он не может сразу вернуть строку (так как может потребоваться некоторое время для загрузки страницы), то возвращает задачу.

Листинг 16.15. Загрузка страницы на основе задач

```
var w = new HttpClient();
string url = "https://endjin.com/";
Task<string> webGetTask = w.GetStringAsync(url);
```

Метод `GetStringAsync` не ожидает завершения загрузки, поэтому возвращается практически сразу. Чтобы выполнить загрузку, компьютер должен отправить сообщение на соответствующий сервер, а затем дождаться ответа.

Когда запрос выполняется, процессор не выполняет никаких действий до тех пор, пока не поступит ответ, а это означает, что большую часть времени этой операции не требуется задействовать поток. Таким образом, этот метод не является оберткой некоторой базовой синхронной версии API в вызове `Task.Run`. На самом деле `HttpClient` не имеет синхронных эквивалентов. В случае классов, которые предлагают API ввода-вывода в обеих формах, таких как `Stream`, синхронные версии часто являются обертками вокруг изначально асинхронной реализации: когда вы вызываете блокирующий API для выполнения ввода-вывода, внутри он обычно выполняет асинхронную операцию, а затем просто блокирует вызывающий поток, пока эта работа не завершится.



Большинство основанных на задачах API придерживаются соглашения об именовании с суффиксом `Async`, и если существует соответствующий синхронный API, он будет иметь то же имя, но без `Async`. Например, класс `Stream` в `System.IO`, который обеспечивает доступ к потокам байтов, имеет метод `Write` для записи байтов в поток, и этот метод является синхронным (т. е. ожидает завершения своей работы перед возвратом). Он также предлагает метод `WriteAsync`. Он делает то же самое, что и `Write`, но, будучи асинхронным, возвращается, не дожидаясь завершения своей работы. Возвращает `Task` для представления работы; это соглашение называется асинхронным шаблоном на основе задач (`TAP`).

Итак, хотя с помощью классов `Task` и `Task<T>` очень легко создавать задачи, которые работают, выполняя методы в потоках пула потоков, они также могут представлять изначально асинхронные операции, которые не требуют использования потока в течение большей части времени своей работы. Хотя это не является частью официальной терминологии, я описывал этот тип операции как беспоточную задачу, чтобы отличать ее от задач, которые полностью выполняются в потоках пула потоков.

ValueTask и ValueTask<T>

`Task` и `Task<T>` являются довольно гибкими, и не только потому, что могут представлять как основанные на потоке, так и беспоточные операции. Как вы увидите, они предлагают несколько механизмов для определения того, когда представляемая ими работа завершается, в том числе предоставляя возможность объединить несколько задач в одну. Несколько потоков могут одновременно ожидать одну и ту же задачу. Вы можете написать механизмы

кэширования, которые повторно раздают одну и ту же задачу, даже спустя долгое время после ее завершения. Это все очень удобно, но вместе с тем означает, что эти типы задач также приводят к дополнительным расходам. Для более ограниченных случаев .NET определяет менее гибкие типы `ValueTask` и `ValueTask<T>`, которые в определенных обстоятельствах оказываются более эффективными.

Наиболее важное различие между этими типами и их обычными аналогами заключается в том, что `ValueTask` и `ValueTask<T>` являются значимыми типами. Это важно в случае чувствительного к производительности кода, поскольку способно уменьшить количество выделяемых кодом объектов и сократить время, которое приложение тратит на работы по сбору мусора. Вы можете подумать, что затраты на переключение контекста, обычно связанные с параллельной работой, скорее всего, окажутся достаточно высоки, поэтому затраты на выделение ресурсов для объектов не будут особенно беспокоить вас при работе с асинхронными операциями. И хотя это часто так и бывает, есть один очень важный сценарий, в котором затраты на сборку мусора для `Task<T>` могут стать проблемой. Это операции, которые иногда выполняются медленно, но обычно нет.

API ввода-вывода очень часто выполняют буферизацию, чтобы уменьшить количество вызовов в ОС. Если нужно записать в поток несколько байтов, они обычно помещают их в буфер и ждут, пока вы не запишете достаточно, чтобы их стоило отправлять в ОС, либо пока вы явно не вызовете `Flush`. Кроме того, буферизация чтения также является обычной практикой — если вы читаете из файла один байт, ОС обычно приходится считывать с диска весь сектор (обычно не менее 4 КБ), и эти данные сохраняются где-то в памяти. Так что при запросе второго байта больше не требуется никаких операций ввода-вывода. На практике результат заключается в том, что если вы пишете цикл, который читает данные из файла относительно небольшими порциями (например, по одной строке текста за раз), большинство операций чтения завершается сразу, поскольку считываемые данные уже были получены ранее.

В случаях, когда подавляющее большинство вызовов в асинхронные API завершаются немедленно, накладные расходы по сборке мусора на создание объектов задач могут оказаться уже значительными. Вот почему в .NET Core 2.0 появились `ValueTask` и `ValueTask<T>`. (Они также доступны в более старых версиях .NET через пакет NuGet `System.Threading.Tasks.Extensions` и являются частью .NET Standard 2.1.) Они позволяют выполнять потенциально асинхронные операции немедленно, без необходимости выделения

каких либо объектов. В случаях, когда немедленное завершение невозможно, эти типы в итоге становятся обертками для объектов `Task` или `Task<T>`, что возвращает дополнительные расходы, но в случаях, когда это необходимо только малой долей вызовов, эти типы могут обеспечить заметный скачок производительности, в частности в коде, который использует методы экономного выделения, описанные в главе 18.

`ValueTask` используется редко, поскольку асинхронные операции, которые не выдают результат, могут просто возвращать статическое свойство `Task`. `CompletedTask`, которое предоставляет собой повторно используемую задачу. Она уже находится в завершенном состоянии, что позволяет избегать затрат на сборку мусора. Но задачи, которые должны произвести результат, как правило, не могут повторно использовать существующие задачи. (Есть некоторые исключения: библиотека классов .NET часто использует предварительно кэшированные уже завершенные задачи для `Task<bool>`, потому что у них есть только два возможных результата. Но в случае `Task<int>` нет разумного способа сохранить список предварительно выполненных задач для каждого возможного результата.)

Эти типы задач значений имеют ряд ограничений. Они одноразовые: в отличие от `Task` и `Task<T>`, вы не должны сохранять эти типы в словаре или `Lazy<T>`, чтобы иметь возможность предоставить кэшированное асинхронное значение. Попытка запросить `Result` у `ValueTask<T>` до завершения — это ошибка. Также ошибкой является и получение `Result` более одного раза. В целом же вы должны использовать `ValueTask` или `ValueTask<T>` только с одной операцией `await` (как описано в главе 17) и не пытаться использовать их еще раз. (С другой стороны, если это необходимо, можно обойти эти ограничения, вызвав метод `AsTask` для получения полной `Task` или `Task<T>` со всеми проистекающими из этого расходами, после чего вам не нужно больше ничего делать с задачей-значением.)

Поскольку задачи типа значения были введены через много лет после появления TPL, библиотеки классов .NET часто используют `Task<T>` там, где вы ожидаете увидеть `ValueTask<T>`. Например, все методы `ReadAsync` класса `Stream` являются отличными кандидатами для этого, но большинство из них были определены задолго до появления `ValueTask<T>`, так что они в основном возвращают `Task<T>`. Недавно добавленная перегрузка, которая принимает `Memory<byte>` вместо `byte[]`, возвращает `ValueTask<T>`. Во многих случаях, когда API были расширены для поддержки новых технологий, эффективно использующих память (и описанных в главе 18), они обычно возвращают

`ValueTask<T>`. А если вы работаете в окружении, чувствительном к производительности, где расходы на сборку мусора на задачу имеют значение, то, скорее всего, подумаете о том, чтобы использовать эти методы в любом случае.

Варианты создания задачи

Вместо использования `Task.Run` можно более глубоко контролировать некоторые аспекты новой задачи на основе потоков, создавая ее с помощью метода `StartNew`, или `Task.Factory`, либо `Task<T>.Factory` в зависимости от того, должна ли ваша задача возвращать результат. Некоторые перегрузки `StartNew` принимают аргумент типа enum `TaskCreationOptions`, который обеспечивает определенный контроль над тем, как TPL планирует задачу.

Флаг `PreferFairness` означает отказ от дружественного к кэшу планирования FIFO, которое обычно используется для пула потоков, и вместо этого ведет к запуску задачи после любых задач, которые уже были запланированы (во многом аналогично устаревшему поведению, которое вы получаете, если используете класс `ThreadPool` напрямую).

Флаг `LongRunning` предупреждает TPL о том, что задача может выполняться долго. По умолчанию планировщик TPL оптимизирует работу с учетом относительно коротких рабочих элементов — до нескольких секунд. Флаг указывает, что работа может занять больше времени, и в этом случае TPL может изменить свое планирование. Если имеется слишком много долгосрочных задач, они могут занять все потоки, и хотя некоторые из рабочих элементов в очереди могут быть предназначены для выполнения более коротких работ, их выполнение все равно займет много времени, поскольку им придется ждать выполнения медленной работы и только потом начать выполнение. Но если TPL знает, какие элементы могут выполняться быстрее, а какие медленнее, он может расставить приоритеты по-другому и избежать подобных проблем.

Другие параметры `TaskCreationOptions` затрагивают отношения между родительскими и дочерними задачами и планировщиками, что я опишу позже.

Статус задачи

За время своего существования задача проходит через несколько состояний, и вы можете использовать свойство `Status` класса `Task`, чтобы узнать текущее. Метод возвращает значение перечисления `TaskStatus`. Если задача успешно завершена, свойство вернет значение перечисления `RanToCompletion`.

Если задача не выполнена, значение будет уже `Faulted`. Если вы отмените задачу, используя метод, показанный в разделе «Отмена» на с. 875, статус будет `Canceled`.

Существует несколько вариаций на тему «в процессе», из которых `Running` является наиболее очевидным — он означает, что какой-то поток в настоящее время выполняет задачу. Задаче, представляющей ввод-вывод, во время ее выполнения обычно не требуется поток, поэтому она никогда не попадает в это состояние. Она запускается в состоянии `WaitingForActivation`, а затем, как правило, переходит непосредственно в одно из трех конечных состояний (`RanToCompletion`, `Faults` или `Canceled`). Задача на основе потоков также может находиться в состоянии `WaitingForActivation`, но только в том случае, если что-то мешает ее выполнению. Обычно это происходит, если вы настроите ее на выполнение только после завершения какой-либо другой задачи (я покажу, как это сделать в самое ближайшее время). Задача, основанная на потоках, также может находиться в состоянии `WaitingToRun`, что означает, что она находится в очереди, ожидающей доступности потока из пула потоков. Есть возможность установить отношения родитель/потомок между задачами, и родитель, который уже завершил работу, но успел создать дочерние задачи (которые еще не завершены), будет находиться в состоянии `WaitingForChildrenToComplete`.

Наконец, есть состояние `Created`. Оно встречается нечасто, потому что представляет собой основанную на потоках задачу, которую вы создали, но еще не попросили запуститься. Задача, созданная с помощью метода `StartNew` фабрики задач или с помощью `Task.Run`, никогда не окажется в этом состоянии, но состояние возможно, если создать новую задачу напрямую.

Уровень детализации в свойстве `TaskStatus` в большинстве случаев не представляет интереса, поэтому класс `Task` определяет различные более простые свойства типа `bool`. Если вы хотите проверить лишь то, что задача больше нечего делать (и не важно, выполнилась она успешно, завершилась сбоем или была отменена), то для этого есть свойство `IsCompleted`. Если вы хотите проверить наличие ошибок или факт отмены, используйте `IsFaulted` или `IsCanceled`.

Получение результата

Предположим, у вас есть `Task<T>`, полученный либо из API, который его предоставляет, либо путем создания задачи на основе потоков, которая возвращает значение. Если задача успешно завершена, вы, очевидно, захотите

получить ее результат. Как и следовало ожидать, его можно извлечь из свойства `Result`. Таким образом, задача, созданная в листинге 16.15, делает содержимое веб-страницы доступным через `webGetTask.Result`.

Если вы попытаетесь прочитать свойство `Result` до завершения задачи, оно заблокирует ваш поток до тех пор, пока результат не станет доступен. (Если у вас есть простая задача, которая не возвращает результат, но вы хотели бы дождаться ее завершения, вы можете вместо этого просто вызвать `Wait`.) Если после этого операция завершается ошибкой, `Result` выдает исключение (как и `Wait`), хотя все не так просто, как можно было бы ожидать. Это я буду обсуждать в разделе «Обработка ошибок» на с. 869.

В большинстве случаев для получения результата гораздо лучше использовать возможности асинхронного языка C#. Это тема следующей главы, но в качестве затравки в листинге 16.16 показано, как можно использовать эту технику для получения результата задачи, которая извлекает веб-страницу. (Понадобится применить ключевое слово `async` перед объявлением метода, чтобы иметь возможность использовать ключевое слово `await`.)

Листинг 16.16. Получение результатов с помощью `await`

```
string pageContent = await webGetTask;
```



Следует избегать использования `Result` в случае незавершенной задачи, так как в ряде случаев это может привести ко взаимоблокировке. Это особенно актуально для настольных приложений, где определенная работа должна выполняться в определенных потоках, так что если вы заблокируете поток, пытаясь прочитать `Result` незавершенной задачи, вы можете помешать ее завершению. Задача может косвенно зависеть от выполнения какой-либо другой работы, а если эту другую работу необходимо выполнить в том же потоке, который вы только что заблокировали, случается взаимоблокировка. Даже если взаимоблокировки нет, блокировка `Result` может вызвать проблемы с производительностью из-за того, что это помешает потокам из пула потоков выполнять какую-то полезную работу. Чтение `Result` незавершенной `ValueTask<T>` не допускается вообще.

Это может показаться не слишком впечатляющим улучшением простой записи `webGetTask.Result`, но, как я покажу в главе 17, данный код не так прост, как кажется. Компилятор C# реструктурирует этот оператор в конечный автомат, управляемый обратным вызовом, который дает возможность

получить результат, не блокируя вызывающий поток. (Если операция не завершена, поток возвращается в место вызова, а оставшаяся часть метода выполняется через некоторое время после завершения операции.)

Но если не использовать функции асинхронного языка, то как можно узнать, что задача завершена? `Result` или `Wait` позволяют вам лишь ожидать, пока это произойдет, блокируя поток, но это, скорее, сводит на нет предназначение асинхронного API в принципе. Возможно, вы захотите получать уведомления о завершении задачи, и этого можно добиться с помощью продолжения.

Продолжения

Задачи содержат различные перегрузки метода с именем `ContinueWith`, который создает новую задачу на основе потоков, выполняющуюся после завершения задачи, для которой был вызван `ContinueWith` (независимо от того, была ли она выполнена успешно, с ошибкой или вообще отменена). В листинге 16.17 эта техника использована для задачи, созданной в листинге 16.15.

Листинг 16.17. Продолжение

```
webGetTask.ContinueWith(t =>
{
    string webContent = t.Result;
    Console.WriteLine("Web page length: " + webContent.Length);
});
```

Задача-продолжение всегда основана на потоке (независимо от того, была ли предшествующая задача основана на потоке, операции ввода-вывода или чем-то еще). Она создается, как только вы вызываете `ContinueWith`, но ее невозможно запустить, пока предшествующая задача не завершится. (Она начинается с состояния `WaitingForActivation`.)



Продолжение – это отдельная задача, и `ContinueWith` возвращает `Task<T>` или `Task` в зависимости от того, возвращает ли результат предоставленный вами делегат. Вы можете задать продолжение для продолжения, если вам требуется связать воедино последовательность операций.

Метод, который вы предоставляете в качестве продолжения (например, лямбда из листинга 16.17), получает задачу-предшественницу в качестве

аргумента, и я использовал этот факт для получения результата. Я мог бы также использовать переменную `webGetTask`, которая находится в области видимости содержащегося метода, поскольку она относится к той же задаче. Однако, используя аргумент, лямбда в листинге 16.17 не использует никаких переменных из содержащего ее метода, что позволяет компилятору генерировать немного более эффективный код — ему не приходится создавать объект для хранения общих переменных, и он может повторно использовать экземпляр делегата, который создает, потому отсутствует необходимость создавать контекстно зависимый экземпляр для каждого вызова. Это означает, что я мог бы с легкостью выделить это в обычный невстроенный метод, если бы заподозрил, что это облегчит чтение кода.

Вы могли подумать, что код из листинга 16.17 чреват возможной проблемой: что, если загрузка завершится настолько быстро, что `webGetTask` завершится прежде, чем коду удастся присоединить продолжение? На самом деле это не имеет значения — если вызвать `ContinueWith` для задачи, которая уже выполнена, она все равно запустит продолжение. Просто это произойдет немедленно. Вы можете присоединить столько продолжений, сколько захотите. Выполнение всех продолжений, которые вы прикрепите до завершения задачи, будет запланировано после ее завершения. И все, что вы прикрепите после выполнения задачи, будет запланировано на немедленное выполнение.

Как и любая другая задача, продолжение по умолчанию будет запланировано для выполнения в пуле потоков. Но есть некоторые способы изменить способ запуска.

Настройки продолжения

Некоторые перегрузки `ContinueWith` принимают аргумент типа перечисления `TaskContinuationOptions`, который управляет тем, как планируется ваша задача (и нужно ли это вообще). В него входят все те же параметры, которые доступны через `TaskCreationOptions`, но добавлен и ряд других, специфичных для продолжений.

Вы можете указать, что продолжение должно выполняться только при определенных обстоятельствах. Например, флаг `OnlyOnRanToCompletion` обеспечит выполнение продолжения только в случае успешного выполнения предшествующей задачи. Флаги `OnlyOnFaults` и `OnlyOnCanceled` имеют очевидное сходное предназначение. В качестве альтернативы вы можете указать `NotOnRanToCompletion`, что означает, что продолжение будет выполняться только в случае сбоя или отмены задачи.



Для одной задачи можно задать несколько продолжений. Из этого следует, что можно задать одно для обработки успешного выполнения, а другое для обработки ошибок.

Вы также можете задать `ExecuteSynchronously`, что будет означать, что продолжение не следует планировать как отдельный рабочий элемент. Обычно, когда задача завершается, любые продолжения этой задачи планируются к выполнению и должны ждать, пока обычные механизмы пула потоков не выберут рабочие элементы из очереди и не выполнят их. (Это не займет много времени, если вы используете параметры по умолчанию и если вы не установите `PreferFairness` — операцию LIFO, которую пул потоков использует для задач, что означает, что самые последние запланированные элементы выполняются первыми.) Однако если ваше продолжение выполняет лишь очень маленький объем работы, расходы на планирование его выполнения в качестве совершенно отдельного элемента могут быть избыточными. Но `ExecuteSynchronously` позволяет объединить задачу завершения с тем же рабочим элементом пула потоков, который выполнил и предшествующую, — TPL запускает такой вид продолжения сразу после завершения предшествующей задачи и перед возвратом потока в пул. Вы должны использовать эту опцию, только если продолжение будет работать быстро.

Настройка `LazyCancellation` отвечает за сложную ситуацию, которая может возникнуть, если вы делаете задачи отменяемыми (как описано далее в «Отмена» на с. 875), но при этом используете продолжения. Если вы отменяете задачу, по умолчанию любые продолжения сразу становятся готовыми к запуску. Но если отменяемая задача сама была настроена как продолжение для другой задачи, которая еще не была завершена, и если она имеет собственное продолжение, как показано в листинге 16.18, то эффект может слегка удивить.

Листинг 16.18. Отмена и цепочка продолжений

```
private static void ShowContinuations()
{
    Task op = Task.Run(DoSomething);
    var cs = new CancellationTokenSource();
    Task onDone = op.ContinueWith(
        _ => Console.WriteLine("Never runs"),
        cs.Token);
```

```
Task andAnotherThing = onDone.ContinueWith(
    _ => Console.WriteLine("Continuation's continuation"));
cs.Cancel();
}

static void DoSomething()
{
    Thread.Sleep(1000);
    Console.WriteLine("Initial task finishing");
}
```

В примере создается задача, которая должна вызвать `DoSomething`, за чем последует отменяемое продолжение для этой задачи (`Task` в `onDone`), а затем заключительная задача, которая служит продолжением для первого продолжения (`andAnotherThing`). Этот код тут же отменяется, что почти наверняка произойдет до того, как будет выполнена первая задача. Результатом будет, что последнее задание выполнится до того, как будет выполнено первое. Последняя задача `andAnotherThing` готова к запуску после завершения `onDone`, даже если это завершение было вызвано отменой `onDone`. Так как здесь была цепочка (`andAnotherThing` — это продолжение для `onDone`, которое, в свою очередь, является продолжением для `op`), то немного странно выглядит то, что `AndAnotherThing` завершает работу до завершения `op`. `LazyCancellation` изменяет поведение так, что первое продолжение не будет считаться выполненным до завершения его предшествующего элемента, что означает, что последнее продолжение будет выполняться только после завершения первой задачи.

Есть и еще один механизм контроля выполнения задач: вы можете задать планировщик.

Планировщики

Все основанные на потоках задачи выполняются `TaskScheduler`. По умолчанию вы получите TPL-планировщик, который запускает рабочие элементы через пул потоков. Однако есть и другие виды планировщиков, и вы можете даже написать собственный.

Наиболее распространенная причина выбора планировщика не по умолчанию заключается в обработке требований привязки потоков. Статический метод `FromCurrentSynchronizationContext` класса `TaskScheduler` возвращает планировщик на основе текущего контекста синхронизации для любого потока, из которого вызывается метод. Данный планировщик выполнит всю

работу именно через этот контекст синхронизации. Таким образом, если вы вызываете `FromCurrentSynchronizationContext` из потока пользовательского интерфейса, результирующий планировщик может использоваться для запуска задач, которые могут безопасно обновлять пользовательский интерфейс. Обычно это используется для продолжений — вы можете запустить какую-то асинхронную работу на основе задач, после чего подключить продолжение, которое обновляет пользовательский интерфейс по завершении этой работы. В листинге 16.19 показана эта техника в применении к файлу с кодом окна в приложении WPF.

При этом для получения планировщика используется инициализатор поля — конструктор для элемента пользовательского интерфейса выполняется в потоке пользовательского интерфейса, поэтому он получит планировщик для контекста синхронизации для потока пользовательского интерфейса. Затем обработчик нажатия мыши загружает веб-страницу с помощью метода `GetStringAsync` класса `HttpClient`. Все выполняется асинхронно, поэтому поток пользовательского интерфейса не блокируется, а это означает, что приложение будет реагировать на запросы во время загрузки. Метод устанавливает продолжение для задачи, используя перегрузку `ContinueWith`, которая принимает `TaskScheduler`. Это гарантирует, что, когда завершается задача, получающая контент, переданная в `ContinueWith` лямбда запускается в потоке пользовательского интерфейса и поэтому имеет безопасный доступ к элементам пользовательского интерфейса.

Листинг 16.19. Планирование продолжения в потоке пользовательского интерфейса

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private static readonly HttpClient w = new HttpClient();
    private readonly TaskScheduler _uiScheduler =
        TaskScheduler.FromCurrentSynchronizationContext();

    private void FetchButtonClicked(object sender, RoutedEventArgs e)
    {
        string url = "https://endjin.com/";
        Task<string> webGetTask = w.GetStringAsync(url);

        webGetTask.ContinueWith(t =>
```

```
{  
    string webContent = t.Result;  
    outputTextBox.Text = webContent;  
},  
_uiScheduler);  
  
}  
}
```

Библиотека классов .NET предоставляет три встроенных вида планировщиков. Имеется планировщик по умолчанию, который использует пул потоков, а также тот, который я только что показал, использующий контекст синхронизации. Третий предоставляется классом `ConcurrentExclusiveSchedulerPair` и, как следует из названия, предоставляет два планировщика, которые доступны через его свойства. Свойство `ConcurrentScheduler` возвращает планировщик, который будет выполнять задачи одновременно, аналогично планировщику по умолчанию. Свойство `ExclusiveScheduler` возвращает планировщик, который можно использовать для запуска задач по одной и временной приостановки работы другого планировщика. (Напоминает семантику синхронизации чтения/записи, которую я описал ранее в этой главе, — она допускает исключительный доступ, когда это необходимо, но в остальное время работает параллельно.)



Хотя это тоже хорошо работает, ключевое слово `await`, описанное в следующей главе, предоставляет более простое решение этой конкретной проблемы.

Обработка ошибок

Объект `Task` указывает, что его работа завершилась сбоем, тем, что входит в состояние `Faulted`. С ошибкой всегда будет связано хотя бы одно исключение, но TPL допускает составные задачи, которые содержат несколько подзадач. Это делает возможным возникновение нескольких сбоев, и корневая задача сообщит обо всех. `Task` определяет свойство `Exception` с типом `AggregateException`. Из главы 8 вы можете помнить, что помимо наследования свойства `InnerException` от базового типа `Exception` `AggregateException` определяет свойство `InnerExceptions`, которое возвращает коллекцию исключений. Здесь вы и найдете полный набор исключений, вызвавших сбой задачи. (Если задача не была составной, там обычно будет одно исключение.)

Если вы попытаетесь получить свойство `Result` или вызвать функцию `Wait` для ошибочной задачи, она вернет тот же `AggregateException`, что можно получить из свойства `Exception`. Сбойная задача отслеживает, использовали ли вы хотя бы один из этих членов, и, если нет, считает исключение незамеченным. TPL использует финализацию для отслеживания сбойных задач с незамеченными исключениями, и, если вы позволите такой задаче стать недостижимой, `TaskScheduler` вызовет собственное статическое событие `UnobservedTaskException`. Это дает вам последний шанс разобраться с исключением, после чего оно будет потеряно.

Пользовательские беспоточные задачи

Многие API на основе ввода/вывода возвращают беспоточные задачи. Если нужно, вы можете сделать то же самое. Класс `TaskCompletionSource<T>` предоставляет способ создать задачу `Task<T>`, у которой нет связанного метода для запуска в пуле потоков, и она вместо этого завершается, когда вы ее об этом просите. Необщенного `TaskCompletionSource` не существует, но он и не нужен. `Task<T>` происходит от `Task`, поэтому вы можете просто выбрать любой тип аргумента. Когда не нужно предоставлять возвращаемое значение, большинство разработчиков по соглашению используют `TaskCompletionSource<object>`.

Предположим, вы используете класс, который не предоставляет API на основе задач, и хотите добавить обертку на основе задач. Класс `SmtpClient`, который я использовал в листинге 16.12, поддерживает более старый асинхронный шаблон на основе событий, а не на основе задач. В листинге 16.20 этот API используется вместе с `TaskCompletionSource<object>` для предоставления основанной на задачах оболочки. (И да, здесь имеется два варианта написания: `Canceled` и `Cancelled`. TPL с постоянством использует `Cancelled`, но более старые API менее разборчивы.)

Листинг 16.20. Использование `TaskCompletionSource<T>`

```
public static class SmtpAsyncExtensions
{
    public static Task SendTaskAsync(this SmtpClient mailClient,
        string from, string recipients, string subject, string body)
    {
        var tcs = new TaskCompletionSource<object>();

        void CompletionHandler(object s, AsyncCompletedEventArgs e)
```

```
    {
        mailClient.SendCompleted -= CompletionHandler;
        if (e.Cancelled)
        {
            tcs.SetCanceled();
        }
        else if (e.Error != null)
        {
            tcs.SetException(e.Error);
        }
        else
        {
            tcs.SetResult(null);
        }
    };

    mailClient.SendCompleted += CompletionHandler;
    mailClient.SendAsync(from, recipients, subject, body, null);

    return tcs.Task;
}
}
```

`SmtpClient` уведомляет нас о завершении операции вызовом события. Прежде всего обработчик этого события отсоединяется (чтобы не запускаться второй раз, если что-то будет использовать тот же `SmtpClient` для дальнейшей работы). Затем определяет, была ли операция выполнена успешно, отменена или завершена с ошибкой, и вызывает метод `SetResult`, `SetCanceled` или `SetException` экземпляра `TaskCompletionSource <object>` соответственно. Это заставит задачу перейти в соответствующее состояние, а также позаботится о запуске любых продолжений, связанных с этой задачей. Источник завершения делает созданный им беспоточный `Task` доступным через свое свойство `Task`, которое возвращается этим методом.

Отношения между родительскими и дочерними задачами

Если метод задачи на основе потоков создает новую подобную задачу, то по умолчанию между ними не будет особой связи. Однако один из флагов `TaskCreationOptions` — это `AttachedToParent`, и если его установить, то вновь созданная задача будет дочерней по отношению к выполняемой в данный момент. Важность этого заключается в том, что родительская задача не будет сообщать о завершении, пока все ее дочерние элементы не будут вы-

полнены. (Конечно, ее собственный метод тоже должен быть завершен.) Если в дочернем элементе возникнет ошибка, родительская задача тоже завершится с ошибкой и включит все исключения дочерних элементов в свой собственный `AggregateException`.



Беспотоковые задачи (например, большинство задач, представляющих ввод-вывод) зачастую нельзя сделать потомками другой задачи. Если вы создаете такую задачу самостоятельно через `TaskCompletionSource<T>`, то это сделать можно, потому что у этого класса есть перегрузка конструктора, которая принимает `TaskCreationOptions`. Однако большинство API .NET, которые возвращают задачи, не предоставляют способа сделать задачу дочерней.

Вы также можете указать флаг `AttachedToParent` для продолжения. Имейте в виду, что это не сделает его дочерним по отношению к предшествующей задаче. Он останется потомком той задачи, которая выполнялась при вызове `ContinueWith` для создания продолжения.

Отношения родитель/потомок — не единственный способ создания задачи, результат которой основан на ряде других элементов.

Составные задачи

Класс `Task` имеет статические методы `WhenAll` и `WhenAny`. У каждого из них есть перегрузки, которые принимают либо коллекцию объектов `Task`, либо коллекцию объектов `Task<T>` в качестве единственного аргумента. Метод `WhenAll` возвращает либо `Task`, либо `Task<T[]>`, которая завершается только тогда, когда все представленные в аргументе задачи завершены (и в последнем случае составная задача создает массив, содержащий результаты каждой отдельной задачи). Метод `WhenAny` возвращает `Task<Task>` или `Task<Task<T>>`, которая завершается вместе с первой задачей, предостав员я ее в качестве результата.

Как и в случае родительской задачи, в случае сбоя любой задачи, составляющей задачу, созданную с помощью `WhenAll`, исключения из всех невыполненных задач будут доступны в исключении `AggregateException` составной задачи. (`WhenAny` не сообщает об ошибках. Завершение происходит, как только первая задача завершается, и вы должны самостоятельно проверить ее на сбой.)

Вы можете прикрепить к этим задачам продолжение, но есть и более непосредственный способ. Вместо создания составной задачи с помощью `WhenAll` или `WhenAny` и последующего вызова метода `ContinueWith` результата можно просто вызвать метод `ContinueWhenAll` или `ContinueWhenAny` фабрики задач. Опять же, они принимают коллекцию `Task` или `Task<T>`, но также и метод для вызова в качестве продолжения.

Другие асинхронные шаблоны

Хотя TPL предоставляет предпочтительный механизм для асинхронных API, .NET существовал почти десять лет, прежде чем такой механизм был добавлен, так что вы столкнетесь и с более старыми подходами. Форма с самой долгой историей – это модель асинхронного программирования (APM). Она была введена в .NET 1.0, поэтому ее можно встретить повсеместно, но ее использование в данный момент не рекомендуется. Согласно этому шаблону, методы объединяются в пары: один для начала работы, а второй – для получения результатов после ее завершения. Листинг 16.21 показывает именно такую пару из класса `Stream` в пространстве имен `System.IO`, а также соответствующий синхронный метод. (Современный код должен вместо этого использовать `WriteAsync` на основе задач.)

Листинг 16.21. Пара APM и соответствующий синхронный метод

```
public virtual IAsyncResult BeginWrite(byte[] buffer, int offset,
                                       int count,
                                       AsyncCallback callback, object state) ...
public virtual void EndWrite(IAsyncResult asyncResult) ...

public abstract void Write(byte[] buffer, int offset, int count) ...
```

Обратите внимание, что первые три аргумента метода `BeginWrite` идентичны аргументам метода `Write`. В APM метод `BeginXxx` принимает все входные данные (т. е. любые обычные аргументы и любые аргументы `ref`, но не аргументы `out`, если таковые присутствуют). Метод `EndXxx` предоставляет любые выходные данные, что означает возвращаемое значение, любые аргументы `ref` (потому что они могут передавать информацию как внутрь, так и наружу) и любые аргументы `out`.

Кроме того, метод `BeginXxx` принимает два дополнительных аргумента: делегат типа `AsyncCallback`, который будет вызываться после завершения

операции, и аргумент типа `object`, который принимает любой объект, который вы бы хотели связать с операцией (или `null`, если вам это не нужно). Метод также возвращает `IAsyncResult`, представляющий асинхронную операцию.

Когда вызывается ваш обратный вызов завершения, вы можете вызвать метод `EndXXX`, передав тот же объект `IAsyncResult`, возвращенный методом `BeginXXX`, и это обеспечит получение возвращаемого значения, если оно есть. Если операция не удалась, метод `EndXXX` вызовет исключение.

Вы можете обернуть API, которые используют APM, в `Task`. Объекты `TaskFactory`, предоставленные `Task` и `Task<T>`, предоставляют методы `FromAsync`, в которые можно передать пару делегатов для методов `BeginXXX` и `EndXXX`, а также любые аргументы, необходимые для метода `BeginXXX`. Результатом будет `Task` или `Task<T>`, представляющий операцию.

Другой распространенный старый шаблон — асинхронный шаблон на основе событий (EAP). Вы уже видели его пример в этой главе — его использует `SmtpClient`. При использовании этого шаблона класс предоставляет метод, который запускает операцию, и соответствующее событие, которое он вызывает, когда операция завершается. Метод и событие обычно имеют связанные имена, например `SendAsync` и `SendCompleted`. Важной особенностью этого шаблона является то, что метод захватывает контекст синхронизации и использует его для вызова события. Это означает, что если вы используете объект, который поддерживает этот шаблон в коде пользовательского интерфейса, он фактически представляет однопоточную асинхронную модель. Это делает его намного проще в использовании, чем APM, потому что вам не нужно писать никакого дополнительного кода, чтобы вернуться в поток пользовательского интерфейса после завершения асинхронной работы.

Не существует автоматизированного механизма обравчивания EAP в задачу, но, как я показал в листинге 16.20, это не особенно-то и сложно.

В асинхронном коде используется еще один распространенный шаблон: шаблон ожидания, поддерживаемый функциями асинхронного языка C# (ключевые слова `async` и `await`). Как я показал в листинге 16.16, этот функционал можно использовать непосредственно с задачами TPL, но язык не распознает `Task` напрямую, так что ожидать можно и что-то отличное от задач. Вы можете использовать ключевое слово `await` со всем, что реализует определенный шаблон. В главе 17 я покажу, как это делается.

Отмена

.NET определяет стандартный механизм отмены медленных операций. Отменяемые операции принимают аргумент типа `CancellationToken`, и если вы установите его в отмененное состояние, операция будет остановлена досрочно (если это возможно).

Сам тип `CancellationToken` не содержит никаких методов для выполнения отмены — API разработан таким образом, чтобы вы могли сообщать операциям, когда вы хотите их отмены, при этом не давая им возможности отменять любые другие операции, которые вы связали с тем же `CancellationToken`. Само действие отмены происходит через отдельный объект, `CancellationTokenSource`. Как следует из названия, вы можете использовать его, чтобы получить любое количество экземпляров `CancellationToken`. Если вы вызываете метод `Cancel` объекта `CancellationTokenSource`, то это переводит все связанные экземпляры `CancellationToken` в состояние отмены.

Некоторым механизмам синхронизации, которые я описал ранее, можно передавать `CancellationToken`. (Происходящие от `WaitHandle` этого не умеют, потому что базовые примитивы Windows не поддерживают модель отмены .NET. `Monitor` также не поддерживает отмену, в отличие от многих новых API.) Также токены отмены обычно принимают API на основе задач, а сам TPL, в свою очередь, предлагает принимающие его перегрузки методов `StartNew` и `ContinueWith`. Если задача уже начала выполняться, TPL ничего не может сделать для ее отмены, но если вы отмените задачу до ее запуска, TPL исключит ее из очереди запланированных задач. Если вам необходима возможность отмены вашей задачи после запуска, вам придется добавить в тело вашей задачи код, который проверяет `CancellationToken` и отменяет работу, если его свойство `IsCancellationRequested` имеет значение `true`.

Поддержка отмены не повсеместна, потому что это не всегда возможно. Некоторые операции просто не могут быть отменены. Например, после отправки сообщения по сети вы не можете его «разотправить». Некоторые операции позволяют отменить работу до достижения определенной точки невозврата. (Если сообщение находится в очереди на отправку, но фактически еще не было отправлено, то, может быть, не слишком поздно его отменить.) Это означает, что даже когда происходит отмена, она может закончиться ничем. Поэтому при использовании отмены следует быть готовым к тому, что она не сработает.

Параллелизм

Библиотека классов .NET включает в себя некоторые классы, которые могут работать с коллекциями данных одновременно в нескольких потоках. Есть три способа сделать это: класс `Parallel`, Parallel LINQ и поток данных TPL.

Класс `Parallel`

Класс `Parallel` содержит три статических метода: `For`, `Foreach` и `Invoke`. Последний из них принимает массив делегатов и выполняет их все по возможности параллельно. (Будет ли использован параллелизм, зависит от различных факторов. В их число входят количество доступных аппаратных потоков, уровень загрузки системы и количество элементов, которые необходимо обработать.) Методы `For` и `Foreach` имитируют конструкции цикла C# с аналогичными именами, но они при этом способны выполнять итерации параллельно.

Листинг 16.22 иллюстрирует использование `Parallel.For` в коде, который выполняет свертку двух наборов выборок. Это очень часто используемая операция, особенно в обработке сигналов. (На практике быстрое преобразование Фурье — это более эффективный способ выполнения этой работы, если только ядро свертки не мало. Однако сложность такого кода затмила бы рассматриваемую тему, класс `Parallel`.) Она создает один выходной элемент на каждый входной. Каждый выходной элемент получается путем вычисления суммы серии пар значений из двух входов, умноженных друг на друга. Для больших наборов данных это может занять много времени, поэтому такую работу можно ускорить, распределив ее по нескольким процессорам. Значение каждого отдельного выходного результата может быть рассчитано независимо от остальных, так что эта задача — хороший кандидат на распараллеливание.

Листинг 16.22. Параллельная свертка

```
static float[] ParallelConvolution(float[] input, float[] kernel)
{
    float[] output = new float[input.Length];
    Parallel.For(0, input.Length, i =>
    {
        float total = 0;
        for (int k = 0; k < Math.Min(kernel.Length, i + 1); ++k)
        {
```

```
        total += input[i - k] * kernel[k];
    }
    output[i] = total;
});

return output;
}
```

Базовая структура этого кода очень похожа на пару вложенных циклов `for`. Я просто заменил внешний цикл `for` вызовом `Parallel.For`. (Я не пытаюсь распараллелить внутренний цикл — если вы делаете каждый отдельный шаг тривиальным, `Parallel.For` будет тратить больше времени на организацию работы, а не на выполнение вашего кода.)

Первый аргумент, `0`, устанавливает начальное значение счетчика цикла, а второй устанавливает верхний предел. Последний аргумент — это делегат, который будет вызываться один раз для каждого значения счетчика цикла. Вызовы будут происходить одновременно, если эвристика класса `Parallel` сообщит ему, что выполняемая параллельно работа способна ускорить получение результата. Запуск этого метода с большими наборами данных на многоядерном компьютере приведет к тому, что все доступные аппаратные потоки будут использоваться на полную мощность.

Возможно, повысить производительность получится, разбив работу на более удобные для кэширования фрагменты, — простое распараллеливание может создать впечатление высокой производительности, задействовав все ядра вашего ЦП, но при этом обеспечивая далеко не оптимальную пропускную способность. Тем не менее всегда присутствует компромисс между сложностью и производительностью, и простота класса `Parallel` часто может показать достойные результаты при относительно небольших затратах.

Parallel LINQ

`Parallel LINQ` — это провайдер LINQ, который работает с информацией в памяти, так же как и `LINQ to Objects`. Пространство имен `System.Linq` делает его доступным как метод расширения `AsParallel`, определенный для любого `IEnumerable<T>` (с помощью класса `ParallelEnumerable`). Он возвращает `ParallelQuery<T>`, который будет поддерживать обычные операторы LINQ.

Любой запрос LINQ, созданный таким образом, предоставляет метод `ForAll`, который принимает делегат. Когда вы его вызываете, он вызывает делегат для всех созданных запросом элементов и если возможно, делает это параллельно в нескольких потоках.

Поток данных TPL

Поток данных TPL – это функция библиотеки классов .NET, которая позволяет создавать граф объектов, выполняющих обработку проходящей через них информации. Вы можете сообщить TPL, какой из узлов должен обрабатывать информацию последовательно, а какие могут работать с несколькими блоками данных одновременно. Вы помещаете данные в граф, а TPL после этого занимается предоставлением каждому узлу блоков на обработку, пытаясь оптимизировать уровень параллелизма в соответствии с доступными вашему компьютеру ресурсами.

API потока данных, который находится в пространстве имен `System.Threading.Tasks.Dataflow` (который вы найдете в пакете NuGet с тем же именем), является большим и сложным, а его описание само по себе может занять целую главу. К сожалению, эта тема выходит за рамки данной книги. Я упомянул этот API лишь потому, что о нем стоит знать при выполнении определенных видов работ.

Итог

Потоки предоставляют возможность выполнять несколько фрагментов кода одновременно. На компьютере с несколькими исполнительными блоками ЦП (т. е. с несколькими аппаратными потоками) вы можете использовать их для параллелизма, задействуя несколько программных потоков. Вы можете явно создавать новые программные потоки с помощью класса `Thread` или же использовать пул потоков или механизм параллелизации, такой как класс `Parallel` или `Parallel LINQ`, чтобы автоматически определять, сколько потоков необходимо для выполнения работы, которую задает ваше приложение. Потоки также способны обеспечить возможность выполнения нескольких одновременных операций, которые не требуют ресурсов ЦП на все время (например, ожидание ответа от внешней службы), но чаще более эффективный способ выполнять такую работу – это асинхронный API (если он доступен). Библиотека параллельных задач (TPL) предоставляет

абстракции, которые полезны для обоих видов параллелизма. Она может управлять несколькими рабочими элементами в пуле потоков с поддержкой объединения нескольких операций и обработки потенциально сложных сценариев ошибок, а его абстракция `Task`, в свою очередь, может представлять изначально асинхронные операции. Если нескольким потокам необходимо использовать и изменять общие структуры данных, вам необходимо использовать механизмы синхронизации, предлагаемые .NET, чтобы гарантировать, что потоки будут правильно координировать свою работу.

ГЛАВА 17

Асинхронные возможности языка

C# обеспечивает поддержку и использование асинхронных методов на уровне языка. Асинхронные API часто являются наиболее эффективным способом использования определенных служб. Например, большинство операций ввода-вывода выполняется асинхронно внутри ядра ОС, поскольку большинство периферийных устройств, таких как контроллеры дисков или сетевые адAPTERы, могут выполнять большую часть своей работы автономно. Им нужно задействовать процессор только в начале и в конце каждой операции.

Хотя многие службы, предлагаемые операционными системами, изначально асинхронны, разработчики часто предпочитают использовать их через синхронные API (т. е. те, которые не возвращаются до завершения работы). Это пустая трата ресурсов, потому что они блокируют поток до завершения ввода-вывода. У потоков есть свои расходы, и для достижения наилучшей производительности всегда лучше иметь относительно небольшое количество потоков ОС. В идеале ваше приложение должно иметь столько же потоков ОС, сколько у вас аппаратных потоков. Но это оптимально, только если вы способны гарантировать, что потоки будут блокироваться, только когда у них не осталось никакой важной работы. (В главе 16 описано различие между потоками ОС и аппаратными потоками.) Чем больше потоков заблокировано в синхронных вызовах API, тем больше потоков потребуется для рабочей нагрузки, что снижает производительность. В чувствительном к производительности коде пригодятся асинхронные API, поскольку вместо того, чтобы тратить ресурсы, заставляя поток ждать завершения ввода-вывода, они дают возможность потоку начать работу, а затем занять его чем-то полезным.

Проблема с асинхронными API состоит в том, что они могут быть значительно более сложными в использовании, особенно если вам нужно координировать множество связанных операций и разбираться с ошибками. Вот почему разработчики часто выбирали менее эффективные синхронные альтернативы в те времена, когда основные языки программирования не

предоставляли встроенной поддержки. В 2012 году такие функции перекочевали из исследовательских лабораторий в C# и VB, и с тех пор многие другие популярные языки получили аналогичные функции (например, JavaScript, получивший очень похожий синтаксис в 2016 году).

Асинхронные функции в C# позволяют писать код, который использует эффективные асинхронные API, при этом в большой мере сохраняя простоту кода, использующего более простые синхронные API.

Асинхронные функции также полезны в некоторых сценариях, при которых максимизация пропускной способности не является основным критерием производительности. При использовании клиентского кода важно избегать блокировки потока пользовательского интерфейса, чтобы сохранить отзывчивость, а асинхронные API предоставляют один из способов сделать это. Языковая поддержка асинхронного кода может помочь разобраться с проблемами привязки потоков, что значительно упрощает работу по написанию отзывчивого кода пользовательского интерфейса.

Ключевые слова `async` и `await`

C# представляет поддержку асинхронного кода через два ключевых слова: `async` и `await`. Первое из них не предназначено для самостоятельного использования. Вы помещаете ключевое слово `async` в объявление метода и этим сообщаете компилятору, что собираетесь использовать в методе асинхронные функции. Если это ключевое слово отсутствует, вы не можете использовать ключевое слово `await`. Возможно, это и излишне — компилятор выдает ошибку, если вы пытаетесь использовать `await` без `async`. Если ему известно, когда тело метода пытается использовать асинхронные функции, зачем это указывать еще и явно? На то есть две причины. Во-первых, как вы увидите далее, эти функции радикально меняют поведение генерированного компилятором кода, поэтому для любого, кто читает код, полезно будет видеть четкое указание на асинхронное поведение метода. Во-вторых, `await` не всегда было ключевым словом C#, поэтому разработчики ранее могли использовать его в качестве идентификатора. Возможно, Microsoft стоило проработать грамматику для `await` так, чтобы слово выступало в качестве ключевого только в очень специфических контекстах, позволяя использовать ее в качестве идентификатора во всех других случаях. Но команда C# решила поступить проще: вы не можете использовать `await` в качестве

идентификатора внутри асинхронного метода, но это допустимый идентификатор во всех остальных местах.

Таким образом, ключевое слово `async` просто объявляет о вашем намерении использовать ключевое слово `await`. (Хотя нельзя использовать `await` без `async`, не будет ошибкой применять ключевое слово `async` к методу, который не использует `await`. Однако это не имеет смысла, поэтому компилятор выдаст предупреждение.) Листинг 17.1 показывает довольно типичное использование. В нем используется класс `HttpClient`, который запрашивает у определенного ресурса только заголовки (используя стандартную команду `HEAD`, которую протокол HTTP определяет как раз для этой цели). Затем он отображает результаты в элементе управления пользовательского интерфейса — этот метод является частью кода для пользовательского интерфейса, который содержит `TextBox` с именем `headerListTextBox`.

Листинг 17.1. Использование `async` и `await` при получении заголовков HTTP

```
// Как вы увидите позже, асинхронные методы обычно не должны быть пустыми
private async void FetchAndShowHeaders(string url, IHttpClientFactory cf)
{
    using (HttpClient w = cf.CreateClient())
    {
        var req = new HttpRequestMessage(HttpMethod.Head, url);
        HttpResponseMessage response =
            await w.SendAsync(req,
                HttpCompletionOption.ResponseHeadersRead);

        headerListTextBox.Text = response.Headers.ToString();
    }
}
```



Ключевое слово `async` не меняет сигнатуру метода. Он определяет, как метод компилируется, а не как используется.

Этот код содержит одно выражение `await`. Ключевое слово `await` нужно использовать в выражении, которое может работать некоторое время до получения результата, и оно указывает, что оставшаяся часть метода не должна выполняться, пока операция не будет завершена. Это очень похоже на то, что делает блокирующий синхронный API, но разница в том, что выражение `await` не блокирует поток, — этот код не так прост, как кажется.

Метод `SendAsync` класса `HttpClient` возвращает `Task<HttpResponseMessage>`, и вам, наверное, уже интересно, почему бы просто не использовать его свойство `Result`. Как вы видели в главе 16, если задача не завершена, это свойство блокирует поток до тех пор, пока результат не станет доступен (или задача завершится со сбоем, и в этом случае она выдаст исключение). Однако в приложении с пользовательским интерфейсом это опасно: если вы блокируете поток пользовательского интерфейса, пытаясь прочитать результат незавершенной задачи, вы остановите выполнение любых операций, которые должны выполняться в этом потоке. Поскольку большая часть работы, выполняемой приложениями пользовательского интерфейса, должна выполняться в потоке пользовательского интерфейса, блокировка этого потока таким образом почти гарантирует, что рано или поздно возникнет взаимоблокировка, что приведет к зависанию приложения. Так что не делайте этого!

Хотя выражение `await` в листинге 17.1 делает что-то, что логически напоминает чтение `Result`, оно работает совсем иначе. Если результат задачи не доступен немедленно, ключевое слово `await` не заставляет поток ждать, несмотря на свое имя. Вместо этого оно вызывает возвращение содержащего метода. Вы можете использовать отладчик, чтобы убедиться, что `FetchAndShowHeaders` возвращается немедленно. Например, если я вызываю этот метод из обработчика события нажатия кнопки, показанного в листинге 17.2, я могу поставить точку останова на вызове `Debug.WriteLine` в этом обработчике, а другую точку останова — на коде в листинге 17.1, который будет обновлять свойство `headerListTextBox.Text`.

Листинг 17.2. Вызов асинхронного метода

```
private void fetchHeadersButton_Click(object sender, RoutedEventArgs e)
{
    FetchAndShowHeaders("https://endjin.com/", this.clientFactory);
    Debug.WriteLine("Method returned");
}
```

Запустив код в отладчике, я обнаружил, что он достигает точки останова в последнем операторе листинга 17.2 прежде, чем точки останова в последнем операторе листинга 17.1. Другими словами, раздел листинга 17.1, который следует за выражением `await`, выполняется после того, как метод вернулся в точку вызова. Очевидно, компилятор каким-то образом организует выполнение оставшейся части метода через обратный вызов, который происходит после завершения асинхронной операции.

Обратите внимание, что код в листинге 17.1 ожидает выполнения в потоке пользовательского интерфейса, поскольку ближе к концу он изменяет свойство `Text` текстового поля. Асинхронные API не обязательно гарантируют, что уведомят вас о завершении в том же потоке, в котором вы начали работу, — более того, большинство этого не сделает. Несмотря на это, листинг 17.1 работает так, как задумано. Так что помимо преобразования половины метода в обратный вызов ключевое слово `await` разбирается и с проблемами привязки потоков.



Отладчик Visual Studio использует кое-какие хитрости при отладке асинхронных методов, что позволяет вам проходить через них, как если бы они были обычными методами. Обычно это полезно, но иногда способно скрыть то, как на самом деле выполняется код. Шаги отладки, которые я только что описал, были придуманы, чтобы пресечь попытки Visual Studio казаться слишком умной, а вместо этого показать, что происходит на самом деле.

Очевидно, что компилятор C# выполняет ряд важных преобразований вавшего кода каждый раз, когда вы используете ключевое слово `await`. В более старых версиях C#, если вам нужно было использовать этот асинхронный API, после чего обновить пользовательский интерфейс, вам пришлось бы написать что-то вроде листинга 17.3. При этом использовалась техника, которую я показал в главе 16, а именно установка продолжения для задачи, возвращаемой `SendAsync`, с использованием `TaskScheduler`, чтобы гарантировать, что тело продолжения выполняется в потоке пользовательского интерфейса.

Листинг 17.3. Написание асинхронного кода вручную

```
private void OldSchoolFetchHeaders(string url, IHttpClientFactory cf)
{
    HttpClient w = cf.CreateClient();
    var req = new HttpRequestMessage(HttpMethod.Head, url);

    var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead)
        .ContinueWith(sendTask =>
    {
        try
        {
            HttpResponseMessage response = sendTask.Result;
```

```
        headerListTextBox.Text = response.Headers.ToString();
    }
    finally
    {
        w.Dispose();
    }
},
uiScheduler);
}
```

Это разумный способ использовать TPL напрямую, и он имеет эффект листинга 17.1, но это не точное представление того, как компилятор C# преобразует код. Как я покажу позже, `await` использует шаблон, который поддерживается, но не требует `Task` или `Task<T>`. Он также генерирует код, который обрабатывает раннее завершение (когда задача уже завершена к тому времени, когда вы готовы к ее ожиданию) гораздо эффективнее, чем в листинге 17.3. Но прежде, чем я перейду к подробностям работы компилятора, сначала хочу проиллюстрировать ряд проблем, которые он за вас решает. Это лучше всего сделать, показав код, который вы могли бы написать до того, как появилась эта языковая функция.

Мой текущий пример довольно прост, потому что включает в себя только одну асинхронную операцию. Но помимо двух шагов, о которых я уже рассказал, — настройка обратного вызова завершения и обеспечение его работы в правильном потоке — мне также пришлось разобраться с оператором `using`, который был в листинге 17.1. Листинг 17.3 не может использовать ключевое слово `using`, потому что мы хотим удалить объект `HttpClient` только после того, как закончим с ним работать. Вызов `Dispose` незадолго до возврата внешнего метода не подойдет, потому что мы должны иметь возможность использовать объект при запуске продолжения, а это обычно происходит чуть позже¹. Поэтому мне нужно создать объект в одном методе (внешнем), а затем избавиться от него в другом методе (вложенном). И поскольку я вызываю `Dispose` вручную, то теперь моя проблема — это исключения, поэтому мне пришлось заключить весь код, который я переместил в обратный вызов, в блок `try` и вызвать `Dispose` в блоке `finally`. (На самом деле я сделал

¹ Этот пример немного надуман, но зато я могу показать, как работает `using` в асинхронных методах. Удаление `HttpClient`, полученного из `IHttpClientFactory`, обычно является необязательным, и в случаях, когда вы создаете новый `HttpClient` напрямую, лучше оставить его и использовать повторно, как описано в разделе «Необязательное удаление» на с. 425.

не все — в маловероятном случае, когда конструктор `HttpRequestMessage` или вызов, который извлекает планировщик задач, вызовет исключение, `HttpClient` не будет удален. Я обрабатываю только тот случай, когда сама операция HTTP завершается с ошибкой.)

В листинге 17.3 для организации продолжения через `SynchronizationContext` использовался планировщик задач, который был текущим на момент начала работы. Это дает гарантию того, что обратный вызов происходит в правильном потоке для обновления пользовательского интерфейса. Ключевое слово `await` способно позаботиться об этом за нас.

Контексты выполнения и синхронизации

Когда выполнение вашей программы достигает выражения `await` для операции, которая не завершается немедленно, код, сгенерированный для этого `await`, обеспечит захват текущего контекста выполнения. (Возможно, ему не придется много трудиться — если это не первое `await` для блокировки в этом методе, и если контекст с тех пор не изменился, он будет уже захвачен.) Когда асинхронная операция завершится, оставшаяся часть вашего метода будет выполнена через контекст выполнения².

Как я описал в главе 16, контекст выполнения имеет дело с определенной контекстной информацией, которая должна передаваться, когда один метод вызывает другой (даже если это происходит косвенно). Но есть и другой вид контекста, способный нас заинтересовать, особенно при написании кода пользовательского интерфейса: контекст синхронизации (который тоже был описан в главе 16).

В то время как все выражения `await` захватывают контекст выполнения, решение о том, следует ли передавать поток контекста синхронизации, также зависит от ожидаемого типа. Если вы ожидаете `Task`, контекст синхронизации также будет захвачен по умолчанию. Задачи — это не единственное, для чего можно использовать `await`, и я опишу, как типы могут поддерживать `await`, в разделе «Шаблон `await`» на с. 899.

Иногда вам может понадобиться избежать использования контекста синхронизации. Если вы хотите выполнить асинхронную работу, начиная с потока пользовательского интерфейса, но у вас нет особой необходимости

² Так получилось, что листинг 17.3 делает так же, потому что TPL захватывает за нас контекст выполнения.

оставаться в этом потоке, планирование каждого продолжения в контексте синхронизации не всегда будет излишней работой. Если асинхронная операция является `Task` или `Task<T>` (или эквивалентные значимые типы, `ValueTask` или `ValueTask<T>`), вы можете объявить, что вы не хотите этого, вызвав метод `ConfigureAwait` и передав ему `false`. Он возвращает другое представление асинхронной операции, и если вы вызовете `await` для него, а не для исходной задачи, текущий `SynchronizationContext`, если он есть, будет проигнорирован. (Не существует эквивалентного механизма отказа от контекста выполнения.) Листинг 17.4 показывает, как использовать эту технику.

Листинг 17.4. `ConfigureAwait`

```
private async void OnFetchButtonClick(object sender, RoutedEventArgs e)
{
    using (HttpClient w = this.clientFactory.CreateClient())
    using (Stream f = File.Create(textBox.Text))
    {
        Task<Stream> getStreamTask = w.GetStreamAsync(urlTextBox.Text);
        Stream getStream = await getStreamTask.ConfigureAwait(false);

        Task copyTask = getStream.CopyToAsync(f);
        await copyTask.ConfigureAwait(false);
    }
}
```

Этот код является обработчиком нажатия кнопки мыши, поэтому изначально выполняется в потоке пользовательского интерфейса. Он извлекает свойство `Text` из пары текстовых полей. Затем он запускает кое-какую асинхронную работу, а именно извлечение содержимого URL-адреса и копирование данных в файл. Он не использует никаких элементов пользовательского интерфейса после получения этих двух свойств `Text`, поэтому не имеет значения, в каком именно потоке выполняется оставшаяся часть метода. Передав `false` в `ConfigureAwait` и ожидая возвращаемого значения, мы сообщаем TPL, что нам все равно, какой поток будет использован, чтобы уведомить нас о завершении. В этом случае это, скорее всего, будет поток из пула потоков. Это позволит выполнять работу эффективнее и быстрее, поскольку позволит избежать ненужного использования потока пользовательского интерфейса после каждого ключевого слова `await`.

Различные асинхронные API, представленные в Windows в качестве части UWP API, возвращают `IAsyncOperation<T>` вместо `Task<T>`. Это связано с тем, что UWP не является ориентированным на .NET и имеет собствен-

ное независимое от времени выполнения представление для асинхронных операций, которое можно использовать еще и в C++ и JavaScript. Этот интерфейс концептуально похож на задачи TPL и поддерживает шаблон `await`, т. е. с этими API вы можете использовать `await`. Тем не менее он не предоставляет `ConfigureAwait`. Если вы хотите написать что-то похожее на листинг 17.4 с использованием одного из этих API, вы можете использовать метод расширения `AsTask`, который оборачивает `IAsyncOperation<T>` в `Task<T>`, и вызвать `ConfigureAwait` уже для этой задачи.



Если вы пишете библиотеки, то в большинстве случаев следует вызывать `ConfigureAwait(false)` везде, где используется `await`. Это связано с тем, что продолжение через контекст синхронизации может быть затратным, а в некоторых случаях способно привести к взаимоблокировке. Единственным исключением являются случаи, когда вы делаете что-то, что определенно требует сохранения контекста синхронизации, или же вы точно знаете, что ваша библиотека всегда будет использоваться только в средах приложений, которые не устанавливают контексты синхронизации. (Например, приложения ASP.NET Core не используют контексты синхронизации, поэтому обычно не имеет значения, вызываете ли вы в них `ConfigureAwait(false)`.)

Листинг 17.1 содержал только одно выражение `await`, но даже это довольно сложно воспроизвести с помощью классического TPL. Листинг 17.4 содержит уже два `await`, и для достижения эквивалентного поведения без ключевого слова `await` потребуется довольно много кода, поскольку исключения могут возникнуть до первого `await`, после второго или между ними, и нам придется вызвать `Dispose` для `HttpClient` и `Stream` в любом из этих случаев (а также в случае, если исключение вообще не выдается). Но все может стать еще сложнее, как только дело вступит управление потоком.

Множественные операции и циклы

Предположим, что вместо получения заголовков или простого копирования тела ответа HTTP в файл мне нужно обработать данные в теле. Если тело большого размера, то его извлечение — это операция, которая может потребовать ряда медленных шагов. Листинг 17.5 шаг за шагом получает веб-страницу.

Листинг 17.5. Несколько асинхронных операций

```
private async void FetchAndShowBody(string url, IHttpClientFactory cf)
{
    using (HttpClient w = cf.CreateClient())
    {
        Stream body = await w.GetStreamAsync(url);
        using (var bodyTextReader = new StreamReader(body))
        {
            while (!bodyTextReader.EndOfStream)
            {
                string line = await bodyTextReader.ReadLineAsync();
                bodyTextBox.AppendText(line);
                bodyTextBox.AppendText(Environment.NewLine);
                await Task.Delay(TimeSpan.FromMilliseconds(10));
            }
        }
    }
}
```

Теперь наш код содержит три выражения `await`. Первое запускает HTTP-запрос `GET`, и операция завершится, когда мы получим первую часть ответа, но сам ответ еще не завершен — далее может следовать несколько мегабайтov контента. Данный код предполагает, что содержимое будет текстовым, поэтому обворачивает возвращаемый объект `Stream` в `StreamReader`, который представляет байты в потоке в виде текста. Затем он использует асинхронный метод `ReadLineAsync` этой обертки для чтения текстовых строк из ответа по одной за раз³. Поскольку данные имеют тенденцию поступать в виде фрагментов, чтение первой строки может занять некоторое время, но следующие несколько вызовов этого метода, вероятно, завершатся немедленно, поскольку каждый полученный нами сетевой пакет будет содержать несколько строк. Но если код способен читать быстрее, чем данные поступают по сети, в конечном итоге он использует все строки из первого пакета и пройдет некоторое время, прежде чем станет доступной следующая строка. Таким образом, вызовы `ReadLineAsync` будут возвращать и задачи, которые выполняются медленно, и задачи, которые выполняются немедленно. Третья асинхронная операция — это вызов `Task.Delay`. Я добавил ее, чтобы замедлить работу и видеть, как данные постепенно поступают в пользовательский интерфейс. `Task.Delay` возвращает задачу, которая за-

³ По правде говоря, следует проверить заголовки ответа HTTP, чтобы выяснить кодировку, и соответствующим образом настроить `StreamReader`. Вместо этого я позволяю ему определить кодировку, что вполне подходит для учебных целей.

вершается после указанной задержки, тем самым обеспечивая асинхронный эквивалент `Thread.Sleep`. (`Thread.Sleep` блокирует вызывающий поток, но `await Task.Delay` добавляет задержку, не блокируя поток.)



Я разместил каждое выражение `await` в отдельном выражении, но это не обязательно. Допустимо писать выражения в форме `(await t1) + (await t2)`. (Можете опустить скобки, если хотите, потому что `await` имеет более высокий приоритет, чем сложение; но мне нравится акцент, который они добавляют.)

Я не собираюсь показывать вам полный доасинхронный эквивалент листинга 17.5, потому что он будет огромным, но опишу некоторые проблемы. Во-первых, у нас есть цикл, в теле которого содержатся два блока `await`. Создание чего-то эквивалентного с помощью `Task` и обратных вызовов будет означать создание собственных элементов цикла, потому что код цикла в конечном итоге разделяется на три метода: тот, который запускает цикл (который будет вложенным методом, действующим в качестве обратного вызова продолжения для `GetStreamAsync`), и два обратных вызова, которые обрабатывают завершение синхронизации `ReadLineA` и `Task.Delay`. Эту проблему можно решить, используя локальный метод, который запускает новую итерацию, вызвав его из двух мест: из точки, в которой вы хотите запустить цикл, и еще раз в продолжении `Task.Delay`, чтобы запустить следующую итерацию. Листинг 17.6 показывает эту технику, но иллюстрирует только одну сторону того, что компилятор сделает за нас; это не полная альтернатива листингу 17.5.

Листинг 17.6. Неполный ручной асинхронный цикл

```
private void IncompleteOldSchoolFetchAndShowBody(
    string url, IHttpClientFactory cf)
{
    HttpClient w = cf.CreateClient();
    var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    w.GetStreamAsync(url).ContinueWith(getStreamTask =>
    {
        Stream body = getStreamTask.Result;
        var bodyTextReader = new StreamReader(body);

        StartNextIteration();
    });
}
```

```
void StartNextIteration()
{
    if (!bodyTextReader.EndOfStream)
    {
        bodyTextReader.ReadLineAsync().ContinueWith(readLineTask =>
        {
            string line = readLineTask.Result;

            bodyTextBox.AppendText(line);
            bodyTextBox.AppendText(Environment.NewLine);

            Task.Delay(TimeSpan.FromMilliseconds(10))
                .ContinueWith(
                    _ => StartNextIteration(), uiScheduler);
        },
        uiScheduler);
    }
},
uiScheduler);
}
```

Этот код кое-как работает, но он даже не пытается утилизировать какие-либо используемые им ресурсы. Есть несколько мест, в которых может произойти сбой, поэтому мы не можем просто добавить один блок `using` или пару `try/finally`, чтобы привести все в порядок. И даже без этого дополнительного усложнения едва ли можно понять, что он пытается выполнить те же базовые операции, что показаны в листинге 17.5. При правильной обработке ошибок он стал бы совершенно нечитаемым. На практике, конечно, было бы проще использовать совершенно другой подход, написав класс, который реализует конечный автомат, позволяющий отслеживать, куда именно попала работа. Это облегчит создание кода, который работает правильно, но едва ли облегчит жизнь кому-то, кто ваш код читает. Будет трудно понять, что то, на что он смотрит, в действительности нечто большее, чем цикл.

Неудивительно, что многие разработчики предпочитали синхронные API. Но C# позволяет писать асинхронный код, который имеет почти такую же структуру, что и его синхронный эквивалент, но предоставляет все преимущества асинхронного кода в плане производительности и быстродействия безо всяких сопутствующих проблем. Если говорить о главном, то это основное преимущество `async` и `await`.

Использование и создание асинхронных последовательностей

В листинге 17.5 показан цикл `while`, но, как и следовало ожидать, вполне допустимо использовать и другие типы циклов, например `for` и `foreach`. Тем не менее `foreach` может добавить головной боли: что произойдет, если коллекция, которую вы перебираете, должна выполнять медленные операции? Это не проблема для типов коллекций, таких как массивы или `HashSet<T>`, все элементы которых уже находятся в памяти, но что насчет `IEnumerable<string>`, возвращаемого `File.ReadLines`? Это очевидный кандидат на асинхронную работу, но на практике он попросту блокирует ваш поток каждый раз, когда ему приходится ждать, пока из хранилища поступят новые данные. А все потому, что шаблон, ожидаемый `foreach`, просто не поддерживает асинхронную работу. Суть проблемы в том, что метод `foreach` будет вызываться для перехода к следующему элементу — он ожидает, что перечислитель (часто, но не всегда являющийся реализацией `IEnumerator<T>`) предоставит метод `MoveNext`, подобный показанному в листинге 17.7.

Листинг 17.7. Совсем не асинхронный `IEnumerable.MoveNext`

```
bool MoveNext();
```

Если ожидается больше элементов, но они пока недоступны, у коллекций нет другого выбора, кроме как блокировать поток, не возвращаясь из `MoveNext` до тех пор, пока данные не поступят. Чтобы это исправить, в C# 8.0 добавили новый шаблон. В библиотеке классов .NET Core 3.0 и .NET Standard 2.1 имеются соответствующие новые типы, которые воплощают этот новый шаблон. Они показаны в листинге 17.8 (впервые представленном в главе 5). Как и в случае синхронного `IEnumerable<T>`, цикл `foreach` не требует использования только этих типов. Подойдет все, что предлагают члены с одинаковой сигнатурой.

Концептуально все идентично синхронному шаблону: асинхронный `foreach` будет запрашивать объект коллекции для перечислителя и постоянно просить его перейти к следующему элементу, выполняя тело цикла со значением, каждый раз возвращаемым `Current` до тех пор, пока перечислитель не укажет, что больше элементов нет. Основное отличие в том, что синхронный метод `MoveNext` был заменен на `MoveNextAsync`, возвращающий `ValueTask<T>`, который можно использовать с `await`. (Интерфейс `IAsyncEnumerable<T>` тоже обеспечивает поддержку передачи токена отмены, хотя асинхронный `foreach` сам не будет это использовать.)

Листинг 17.8. `IAsyncEnumerable<T>` и `IAsyncEnumerator<T>`

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(
        CancellationToken cancellationToken = default);
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }

    ValueTask<bool> MoveNextAsync();
}
```

Чтобы использовать перечислимый источник, который реализует данный шаблон, нужно поместить ключевое слово `await` перед `foreach`. С# также помогает реализовать этот шаблон: в главе 5 показано, как можно использовать ключевое слово `yield` в методе итератора для реализации `IEnumerable<T>`, но вы также можете вернуть и `IAsyncEnumerable <T>`.

Листинг 17.9 показывает как реализацию, так и использование `IAsyncEnumerable<T>`.

Листинг 17.9. Использование и создание асинхронных перечислителей

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;

namespace AsyncEnum
{
    internal static class Program
    {
        private static async Task Main(string[] args)
        {
            await foreach (string line in ReadLinesAsync(args[0]))
            {
                Console.WriteLine(line);
            }
        }

        private static async IAsyncEnumerable<string>
            ReadLinesAsync(string path)
        {
```

```

        using (var bodyTextReader = new StreamReader(path))
    {
        while (!bodyTextReader.EndOfStream)
        {
            string line = await bodyTextReader.ReadLineAsync();
            yield return line;
        }
    }
}

```



Как показывает этот пример, можно сделать метод `Main` программы на C# асинхронным. Вы должны вернуть либо `Task`, либо `Task<int>`. (Последний вариант позволяет, если нужно, генерировать ненулевой код выхода.) Среда выполнения .NET не поддерживает асинхронные точки входа, поэтому компилятор C# генерирует скрытый метод, который действует как реальная точка входа, вызывая асинхронный `Main`, а затем блокируется, пока задача, которую тот возвращает, не завершится.

Поскольку эта языковая поддержка делает создание и использование `IAsyncEnumerable<T>` очень похожими на работу с `IEnumerable<T>`, вам может быть интересно узнать, существуют ли асинхронные версии различных операторов LINQ, описанных в главе 10. В отличие от `LINQ to Objects`, реализации `IAsyncEnumerable<T>` располагаются не в частях библиотеки классов, встроенной в .NET или .NET Standard, но в предоставленном Microsoft соответствующем пакете NuGet. Если вы добавите ссылку на пакет `System.Linq.Async` и добавите в начало кода `using System.Linq;`, то все операторы LINQ станут доступны для выражений `IAsyncEnumerable<T>`.

Разбирая асинхронные эквиваленты широко используемых типов, мы должны взглянуть и на `IAsyncDisposable`.

Асинхронное удаление

Как описано в главе 7, интерфейс `IDisposable` реализуется типами, которым необходимо оперативно выполнять какую-либо очистку, например закрытие открытого дескриптора, и для которых существует языковая поддержка в форме операторов `using`. Но что, если очистка включает в себя потенциально медленную работу, такую какброс данных на диск? С учетом такого сценария в .NET Core 3.0 и .NET Standard 2.1 был добавлен новый интерфейс

`IAsyncDisposable`. Как показано в листинге 17.10, в C# 8.0 добавлена его поддержка: вы можете поместить ключевое слово `await` перед оператором `using`, чтобы использовать асинхронно удаляемый ресурс. (Вы также можете поместить `await` перед объявлением `using`.)

Листинг 17.10. Использование и реализация `IAsyncDisposable`

```
using System;
using System.IO;
using System.Threading.Tasks;

namespace AsyncDispose
{
    class Program
    {
        static async Task Main(string[] args)
        {
            await using (var w = new DiagnosticWriter(@"c:\temp\log.txt"))
            {
                await w.LogAsync("Test");
            }
        }
    }

    class DiagnosticWriter : IAsyncDisposable
    {
        private StreamWriter fs;

        public DiagnosticWriter(string path)
        {
            fs = new StreamWriter(path);
        }

        public Task LogAsync(string message) =>
            fs.WriteLineAsync(message);

        public async ValueTask DisposeAsync()
        {
            if (fs != null)
            {
                await fs.FlushAsync();
                fs = null;
            }
        }
    }
}
```



Хотя ключевое слово `await` располагается перед оператором `using`, ожидаемая потенциально медленная операция происходит, когда выполнение покидает блок оператора `using`. Это неизбежно, поскольку операторы `using` и объявления по сути скрывают вызов `Dispose`.

В листинге 17.10 также показано, как реализовать `IAsyncDisposable`. В то время как синхронный `IDisposable` определяет единственный метод `Dispose`, его асинхронный аналог определяет единственный метод `DisposeAsync`, который возвращает `ValueTask`. Это позволяет нам аннотировать метод с помощью `async`. Оператор `async using` гарантирует, что задача, возвращаемая `DisposeAsync`, завершится в конце своего блока до продолжения выполнения. Возможно, вы заметили, что мы использовали несколько различных типов возврата для методов, использующих `async`. Итераторы — это особый случай (так же, как и в синхронном коде), но как насчет методов, которые возвращают различные типы задач?

Возвращение задачи

Любой метод, который использует `await`, сам по себе способен занять определенное время для запуска, поэтому помимо возможности использования асинхронных API нужно представить асинхронный публичный фасад. Компилятор C# позволяет методам, помеченным ключевым словом `async`, возвращать объект, который представляет асинхронную работу в процессе выполнения. Вместо возврата `void` можно вернуть `Task` или `Task<T>`, где `T` — это любой тип. Это дает вызывающей стороне возможность проверить состояние работы, выполняемой вашим методом, возможность прикрепить продолжения и, если вы используете `Task<T>`, способ получить результат. Кроме того, вы можете вернуть эквивалентные значимые типы — `ValueTask` и `ValueTask<T>`. Возврат любого из них означает, что если ваш метод вызывается из другого асинхронного метода, он может использовать `await` для ожидания завершения вашего метода и, если возможно, для получения его результата.

Возврат задачи почти всегда предпочтительнее `void`, потому что в случае `void` вызывающая сторона не сможет узнать, когда ваш метод действительно завершился или когда он вызвал исключение. (Асинхронные методы способны продолжать выполняться после возврата (вообще, в этом-то и весь смысл!), поэтому к тому времени, когда вы создадите исключение, исходный вызывающий объект, вероятно, будет уже не в стеке.) Возвращая объект задачи,

вы предоставляете компилятору способ сделать исключения доступными и, где это возможно, способ предоставить результат.

Возврат задачи настолько прост, что нет причин его не делать. Чтобы изменить метод в листинге 17.5 с учетом возврата задачи, мне нужно внести всего лишь одно изменение. Я делаю возвращаемым типом `Task` вместо `void`, как показано в листинге 17.11, а остальную часть кода можно оставить нетронутой.

Листинг 17.11. Возвращение задачи

```
private async Task FetchAndShowBody(string url, IHttpClientFactory cf)  
// ... тот же самый код
```

Компилятор автоматически генерирует код, необходимый для создания объекта `Task` или `ValueTask`, и устанавливает его в завершенное или ошибочное состояние, когда метод возвращается или выдает исключение. А если вы хотите вернуть результат своей задачи, то это тоже очень просто. Просто сделайте возвращаемым типом `Task<T>` или `ValueTask<T>`, после чего используйте ключевое слово `return`, как если бы тип возврата вашего метода был просто `T`, как это показано в листинге 17.12.

Листинг 17.12. Возврат `Task<T>`

```
public static async Task<string> GetServerHeader(  
    string url, IHttpClientFactory cf)  
{  
  
    using (HttpClient w = cf.CreateClient())  
  
    {  
  
        var request = new HttpRequestMessage(HttpMethod.Head, url);  
        HttpResponseMessage response = await w.SendAsync(  
            request, HttpCompletionOption.ResponseHeadersRead);  
  
        string result = null;  
        IEnumerable<string> values;  
        if (response.Headers.TryGetValues("Server", out values))  
        {  
            result = values.FirstOrDefault();  
        }  
        return result;  
    }  
}
```

Код асинхронно извлекает заголовки HTTP, как и в листинге 17.1, но вместо отображения результатов выбирает значение первого заголовка `Server`: и делает его результатом `Task<string>`, которую этот метод и возвращает. Как видите, оператор `return` возвращает простую строку, хотя тип возврата метода — `Task<string>`. Компилятор генерирует код, который завершает задачу, и организует все так, чтобы эта строка стала результатом. В случае возвращаемого типа `Task` или `Task<T>` сгенерированный код создает задачу, аналогичную той, что вы получите с использованием `TaskCompletionSource<T>`, описанным в главе 16.



Так же как ключевое слово `await` может использовать любой асинхронный метод, который соответствует определенному шаблону (см. далее), C# предлагает такую же гибкость в применении к реализации асинхронного метода. Вы не ограничены `Task`, `Task<T>`, `ValueTask` и `ValueTask<T>`. Вы можете вернуть любой тип, который удовлетворяет двум условиям: он должен быть аннотирован атрибутом `AsyncMethodBuilder`, определяющим класс, который компилятор может использовать для управления ходом выполнения и завершением задачи, а также содержать метод `GetAwaiter`, который возвращает тип, реализующий интерфейс `ICriticalNotifyCompletion`.

В возврате одного из встроенных типов задач почти нет минусов. Вызывающие процедуры не обязаны что-либо с ним делать, поэтому ваш метод будет так же прост в использовании, как и метод с `void`, но с дополнительным преимуществом в том, что задача доступна для вызывающей стороны, которой она требуется. Единственной причиной возврата `void` может быть то, что некое внешнее ограничение заставляет ваш метод иметь определенную сигнатуру. Например, большинство обработчиков событий должны иметь в качестве возвращаемого типа `void`. Но если вы не обязаны его использовать, `void` — это не рекомендуемый тип возврата для асинхронного метода.

Применение `async` ко вложенным методам

В приведенных выше примерах я применял ключевое слово `async` к обычным методам. Его можно использовать в анонимных функциях (методах или лямбдах), а также в локальных функциях. Например, если вы пишете программу, которая создает элементы пользовательского интерфейса про-

граммным способом, может показаться удобным подключать обработчики событий, написанные в виде лямбда-выражений, и вы захотите сделать некоторые из них асинхронными, как в листинге 17.13.

Листинг 17.13. Асинхронная лямбда

```
okButton.Click += async (s, e) =>
{
    using (HttpClient w = this.clientFactory.CreateClient())
    {
        infoTextBlock.Text = await w.GetStringAsync(uriTextBox.Text);
    }
};
```



Это не имеет ничего общего с асинхронным вызовом делегата — устаревшей техникой, о которой я упоминал в главе 9 и которая широко использовалась для работы с пулом потоков до того, как анонимные методы и TPL предоставили лучшую альтернативу.

Шаблон await

Большинство асинхронных API, поддерживающих ключевое слово `await`, будут возвращать задачи TPL определенного вида. Но это не категоричное требование C#. `Await` будет работать со всем, что реализует определенный шаблон. (Так, приложения UWP могут использовать `await`, даже несмотря на то, что API в этой среде не возвращают задачи TPL.) Более того, хотя `Task` и поддерживает этот шаблон, его работа означает, что компилятор использует задачи немного по-другому, чем это делали бы вы при непосредственном использовании TPL. Отчасти поэтому я ранее говорил, что код, показывающий основанные на задачах асинхронные эквиваленты кода на основе `await`, не точно отражают то, что делает компилятор. В этом разделе я собираюсь показать, как компилятор использует задачи и другие типы, которые поддерживают `await`, с целью как можно лучше проиллюстрировать, как же он работает на самом деле.

Я создам собственную реализацию шаблона `await`, чтобы показать, чего ожидает компилятор C#. В листинге 17.14 показан асинхронный метод `UseCustomAsync`, который использует эту пользовательскую реализацию. Он помещает результат выражения `await` в `string`, следовательно, он явно ожидает, что асинхронная операция выдаст строку в качестве вывода. В при-

мере вызывается метод `CustomAsync`, который возвращает эту реализацию шаблона. Как видите, это не `Task<string>`.

Листинг 17.14. Вызов пользовательской реализации `await`

```
static async Task UseCustomAsync()
{
    string result = await CustomAsync();
    Console.WriteLine(result);
}

public static MyAwaitableType CustomAsync()
{
    return new MyAwaitableType();
}
```

Компилятор ожидает, что операнд ключевого слова `await` будет типом, который содержит метод с именем `GetAwaiter`. Это может быть обычный член экземпляра или метод расширения. (Таким образом, можно заставить `await` работать с типом, который изначально его не поддерживает, определив подходящий метод расширения.) Этот метод должен возвращать объект или значение, которое называется *ждущим* и выполняет три действия.

Во-первых, ждущий объект должен предоставить свойство типа `bool` с именем `IsCompleted`. Код, сгенерированный компилятором для `await`, использует его, чтобы определить, была ли операция уже завершена. В ситуациях, когда нет необходимости выполнять медленную работу (например, когда вызов `ReadAsync` в потоке может быть обработан немедленно с данными, которые поток уже имеет в буфере), задание обратного вызова будет бесполезной работой. Поэтому `await` избегает создания ненужного делегата, если свойство `IsCompleted` возвращает `true`, и просто продолжает выполнение остальной части метода.

Компилятору также нужен способ получить результат после завершения работы, поэтому у ожидающего должен быть метод `GetResult`. Его возвращаемый тип определяет тип результата операции — это будет тип выражения `await`. (Если результата нет, тип возвращаемого значения будет `void`. `GetResult` все равно нужен, потому что он отвечает за выдачу исключений в случае сбоя.) Поскольку в листинге 17.14 результат `await` присваивается переменной типа `string`, метод `GetResult` ждущего объекта, возвращаемого методом `GetAwaiter` класса `MyAwaitableType`, должен быть `string` (или типом, неявно преобразуемым в `string`).

Наконец, компилятор должен быть в состоянии предоставить обратный вызов. Если `IsCompleted` возвращает `false`, указывая на то, что операция еще не завершена, то код, сгенерированный для выражения `await`, создаст делегат, который будет выполнять остальную часть метода. Компилятору нужна возможность передать его ожидающему. (Это похоже на передачу делегата в метод задачи `ContinueWith`.) Для этого компилятору нужен не только метод, но и интерфейс. Вам придется реализовать `INotifyCompletion`, а еще есть дополнительный интерфейс, который тоже рекомендуется по возможности реализовать. Он называется `ICriticalNotifyCompletion`. Они выполняют похожие задачи: каждый определяет один метод (`OnCompleted` и `UnsafeOnCompleted` соответственно), который принимает единственный делегат `Action`, и ожидающий должен вызывать этот делегат после завершения операции. Различие между этими интерфейсами и их соответствующими методами заключается в том, что первый требует, чтобы ожидающий передавал текущий контекст выполнения целевому методу, а последний — нет. Функции библиотеки классов .NET, которые компилятор C# использует для создания асинхронных методов, всегда передают контекст выполнения за вас, поэтому там, где это доступно, сгенерированный код обычно вызывает `UnsafeOnCompleted`, чтобы избежать его повторной передачи. (Если компилятор использует `OnCompleted`, ожидающая сторона тоже будет передавать контекст.) Однако в случае .NET Framework вы обнаружите, что ограничения безопасности способны препятствовать использованию `UnsafeOnCompleted`. (В .NET Framework была концепция ненадежного кода. На код из потенциально ненадежного источника — например, загруженный из интернета — будут наложены различные ограничения. От этой концепции отказались в .NET Core, но от нее остались следы, такие как эта деталь работы асинхронных операций.) Поскольку `UnsafeOnCompleted` не передает контекст выполнения, ненадежному коду не разрешается его вызывать, потому что такой вызов даст способ обойти определенные механизмы безопасности. Реализации `UnsafeOnCompleted` в .NET Framework предstawляемые для различных типов задач, помечены `SecurityCriticalAttribute`, что означает, что метод может вызывать только полностью надежный код. Нам же нужен `OnCompleted`, чтобы не полностью надежный код мог использовать ждущий объект.

Листинг 17.15 показывает минимальную рабочую реализацию шаблона `await`. Она чрезмерно упрощена, так как всегда завершается синхронно, поэтому ее метод `OnCompleted` ничего не делает. Фактически при использовании шаблона `await` этот метод никогда не будет вызываться, поэтому у меня он

вызывает исключение. И хотя этот пример крайне прост, он служит прекрасной иллюстрацией того, что делает `await`.

Листинг 17.15. Простейшая реализация шаблона `await`

```
public class MyAwaitableType
{
    public MinimalAwaiter GetAwaiter()
    {
        return new MinimalAwaiter();
    }

    public class MinimalAwaiter : INotifyCompletion
    {
        public bool IsCompleted => true;

        public string GetResult() => "This is a result";

        public void OnCompleted(Action continuation)
        {
            throw new NotImplementedException();
        }
    }
}
```

Видя этот код, мы теперь знаем, что будет делать листинг 17.14. Он вызовет `GetAwaiter` экземпляра `MyAwaitableType`, возвращаемого методом `CustomAsync`. Затем он проверит свойство `IsCompleted` ожидающего, и если оно равно `true` (а так и будет), он немедленно запустит остальную часть метода. Компилятор не знает, что `IsCompleted` в данном случае всегда будет равен `true`, поэтому он генерирует код для обработки случая с `false`. Он создаст делегат, который при вызове запустит остальную часть метода, и передаст этот делегат методу `OnCompleted` ожидающего. (Я не добавил `UnsafeOnCompleted`, поэтому он вынужден будет использовать `OnCompleted`.) В листинге 17.16 показан код, который все это делает.

Листинг 17.16. Очень грубое отражение того, что делает `await`

```
static void ManualUseCustomAsync()
{
    var awaier = CustomAsync().GetAwaiter();
    if (awaier.IsCompleted)
    {
        TheRest(awaier);
    }
}
```

```
        else
    {
        awaiter.OnCompleted(() => TheRest(awaiter));
    }
}

private static void TheRest(MyAwaitableType.MinimalAwaiter awaiter)
{
    string result = awaiter.GetResult();
    Console.WriteLine(result);
}
```

Я разделил метод на две части, потому что компилятор C# избегает создания делегата в случае, когда `IsCompleted` имеет значение `true`, и я хотел сделать то же самое. Однако это не совсем то, что делает компилятор C# — ему также удается избежать создания дополнительного метода для каждого оператора `await`, а это означает, что ему приходится создавать значительно более сложный код. Фактически в случае методов, которые содержат лишь один `await`, он делает гораздо больше работы, чем листинг 17.16. Но, как только число выражений `await` начинает расти, сложность окупается, поскольку компилятору не нужно добавлять дополнительные методы. Листинг 17.17 показывает нечто более близкое к тому, что делает компилятор.

Листинг 17.17. Немного более точное отражение того, как работает `await`

```
private class ManualUseCustomAsyncState
{
    private int state;
    private MyAwaitableType.MinimalAwaiter awaiter;

    public void MoveNext()
    {
        if (state == 0)
        {
            awaiter = CustomAsync().GetAwaiter();
            if (!awaiter.IsCompleted)
            {
                state = 1;
                awaiter.OnCompleted(MoveNext);
                return;
            }
        }
        string result = awaiter.GetResult();
        Console.WriteLine(result);
    }
}
```

```
}

static void ManualUseCustomAsync()
{
    var s = new ManualUseCustomAsyncState();
    s.MoveNext();
}
```

Это все еще проще, чем реальный код, но основная стратегия налицо: компилятор генерирует вложенный тип, который действует как конечный автомат. У него есть поле (`state`), которое показывает, до какой точки дошел метод, и он же содержит поля, соответствующие локальным переменным метода. (В этом примере есть лишь переменная `awaiter`.) Когда асинхронная операция не блокируется (т. е. ее `IsCompleted` немедленно возвращает `true`), метод может просто перейти к следующей части. Но как только он встречает операцию, которая требует времени, он обновляет переменную `state`, чтобы запомнить свое местоположение, а затем использует метод `OnCompleted` соответствующего ожидающего. Обратите внимание, что метод, который запрашивается при завершении, тот же, что уже запущен: `MoveNext`. И это так вне зависимости от того, сколько операторов `await` нужно выполнить, — каждый обратный вызов завершения вызывает один и тот же метод, класс просто запоминает, как далеко он уже прошел, и метод продолжает оттуда.

Не буду показывать реальный сгенерированный код. Он практически нечитаем, потому что содержит множество непередаваемых идентификаторов. (Из главы 3 вы помните, что когда компилятору C# необходимо генерировать элементы с идентификаторами, которые не должны конфликтовать с нашим кодом или вообще быть для него видимыми, он создает имя, которое среда выполнения считает допустимым, но оно недопустимо в C#; это называется непередаваемым именем.) Кроме того, сгенерированный компилятором код использует различные вспомогательные классы из пространства имен `System.Runtime.CompilerServices`, которые предназначены для использования только из асинхронных методов и для управления такими вещами, как определение интерфейса завершения ожидающего и обработка передачи связанного контекста выполнения. Кроме того, если метод возвращает задачу, имеются дополнительные вспомогательные методы для ее создания и обновления. Но если требуется понимание природы взаимосвязи типа, допускающего `await`, и кода, который компилятор создает для выражения `await`, листинг 17.17 будет как нельзя кстати.

Обработка ошибок

Ключевое слово `await` работает с исключениями так, как и следовало ожидать: в случае сбоя асинхронной операции исключение возникает в выражении `await`, которое использовало эту операцию. Общий принцип, согласно которому асинхронный код может быть структурирован так же, как обычный синхронный, продолжает применяться и в условиях исключений, и компилятор делает все возможное для того, чтобы это было так.

Листинг 17.18 содержит две асинхронные операции, одна из которых проходит в цикле. Этим он напоминает листинг 17.5. С получаемым содержимым он делает нечто другое, но главное в том, что он возвращает задачу. Это хорошее место для возникновения ошибки, если какая-либо из операций потерпит неудачу.

Листинг 17.18. Несколько потенциальных точек отказа

```
private static async Task<string> FindLongestLineAsync(
    string url, IHttpClientFactory cf)
{
    using (HttpClient w = cf.CreateClient())
    {
        Stream body = await w.GetStreamAsync(url);
        using (var bodyTextReader = new StreamReader(body))
        {
            string longestLine = string.Empty;
            while (!bodyTextReader.EndOfStream)
            {
                string line = await bodyTextReader.ReadLineAsync();
                if (longestLine.Length > line.Length) {
                    longestLine = line;
                }
            }
            return longestLine;
        }
    }
}
```

Исключения могут вызывать трудности в случае асинхронных операций, потому что к тому времени, когда произойдет сбой, наиболее вероятно, что метод, который начал работу, уже вернется. Метод `FindLongestLineAsync` в этом примере, как правило, будет возвращаться, как только выполнит первое выражение `await`. (Возможно, этого и не произойдет. Если соответствующий ресурс находится в локальном HTTP-кэше или если `IHttpClientFactory`

возвращает клиент, настроенный как фальшивый и никогда не выполняющий никаких реальных запросов, то эта операция может сразу завершиться успешно. Но обычно эта операция занимает некоторое время, вызывая возврат метода.) Предположим, что операция выполнена успешно и начинает выполняться остальная часть метода, но на полпути, в цикле, который извлекает тело ответа, компьютер теряет сетевое подключение. Это приведет к сбою одной из операций, запущенных `ReadLineAsync`.

Исключение возникнет из `await` для этой операции. Но в этом методе нет обработки исключений. Так что же будет дальше? Обычно мы ожидаем, что исключение начнет передаваться вверх по стеку, но что расположено в стеке над этим методом? Это почти наверняка не будет код, который первоначально его вызывал, — вспомните, что метод обычно возвращаетя, как только достигает первого `await`, так что на этом этапе мы работаем в результате обратного вызова, который ожидающий произвел для задачи, возвращенной `ReadLineAsync`. Скорее всего, мы будем работать в каком-то потоке из пула потоков, а код, который находится прямо над нами в стеке, будет частью ожидающего выполнения задачи. И он не будет знать, что делать с нашим исключением.

Но исключение не распространяется вверх по стеку. Когда исключение оказывается необработанным в асинхронном методе, который возвращает задачу, сгенерированный компилятором код перехватывает его и переводит задачу, возвращенную этим методом, в состояние ошибки (что, в свою очередь, будет означать, что все, кто ожидал эту задачу, теперь могут продолжать работу). Если код, который вызвал `FindLongestLineAsync`, работает напрямую с TPL, он сможет увидеть исключение, обнаружив это состояние ошибки и получив свойство `Exception` задачи. В качестве альтернативы он может либо вызвать `Wait`, либо получить свойство `Result` задачи, и в обоих случаях задача генерирует исключение `AggregateException`, содержащее исходное исключение. Но если код, вызывающий `FindLongestLineAsync`, использует `await` для задачи, которую мы возвращаем, исключение будет повторно вызвано оттуда. С точки зрения вызывающего кода это выглядит так, как будто исключение возникло обычным образом, что показано в листинге 17.19.

Код выглядит обманчиво простым. Помните, что компилятор выполняет существенную реструктуризацию кода вокруг каждого `await` и выполнение того, что выглядит как один метод, может на практике превратиться в несколько вызовов. Таким образом, сохранение семантики даже такого простого

блока обработки исключений, как этот (или связанных концепций, таких как оператор `using`), нетривиально. Если вы когда-либо пытались написать эквивалентную обработку ошибок в асинхронной работе без помощи компилятора, вы оцените, как много C# делает здесь за вас.

Листинг 17.19. Обработка исключений из await

```
try
{
    string longest = await FindLongestLineAsync(
        "http://192.168.22.1/", this.clientFactory);
    Console.WriteLine("Longest line: " + longest);
}
catch (HttpRequestException x)
{
    Console.WriteLine("Error fetching page: " + x.Message);
}
```



Ожидание не будет повторно вызывать `AggregateException`, предоставленное свойством `Exception` задачи. Повторно будет выброшено оригинальное исключение. Это позволяет асинхронным методам обрабатывать ошибку так же, как это делает синхронный код.

Проверка аргументов

Один аспект того, как C# автоматически сообщает об исключениях через задачу, которую возвращает ваш асинхронный метод, может вас удивить. Это будет означать, что код, подобный листингу 17.20, будет делать не совсем то, что вы от него ожидаете.

Листинг 17.20. Теоретически неожиданная проверка аргументов

```
public async Task<string> FindLongestLineAsync(string url)
{
    if (url == null)
    {
        throw new ArgumentNullException("url");
    }
    ...
}
```

Внутри асинхронного метода компилятор обрабатывает все исключения одинаково: ни одному из них не разрешается пренебрегать стеком, как это было бы в случае обычного метода, и о них всегда будет сообщено при

сбое возвращенной задачи. Это верно даже для исключений, выданных до первого `await`. В этом примере проверка аргумента происходит до того, как метод успевает сделать что-либо еще, поэтому на данном этапе мы все еще будем работать в потоке первоначального вызывающего кода. Возможно, вы думали, что исключение аргумента, выданное этой частью кода, будет передаваться непосредственно вызывающей стороне. Фактически вызывающая сторона увидит возврат без исключения, создав задачу, которая находится в состоянии ошибки.



Если невозможно без медленной работы определить, является ли конкретный аргумент допустимым, вы не сможете следовать этому соглашению, если вам нужен действительно асинхронный метод. В этом случае вам следует решить, нужна ли вам блокировка метода до тех пор, пока он не сможет проверить все аргументы, или достаточно сообщить об исключениях аргументов через возвращенную задачу вместо их немедленного вызова.

Если вызывающий метод немедленно вызовет `await` в задаче возврата, это не будет иметь особого значения — в любом случае он увидит исключение. Но есть код, который не будет ждать сразу, и в этом случае он до определенного времени не увидит исключения. Общепринятым соглашением для простых исключений проверки аргументов является то, что, если вызывающий код допустил явную программную ошибку, мы должны немедленно сгенерировать исключение, но показанный код этого не делает.

В случаях, когда нужно немедленно выбросить такое исключение (допустим, оно вызывается из кода, который не ожидает немедленного результата, а вы хотите узнать о проблеме как можно скорее), обычная техника — написать обычный метод, который проверяет аргументы перед вызовом выполняющего работу асинхронного метода, и сделать этот второй метод закрытым или локальным. (Между прочим, вам придется делать нечто подобное, если нужна немедленная проверка аргументов в случае итераторов. Итераторы были описаны в главе 5.) Листинг 17.21 показывает такой публичный метод-обертку и запуск метода, который он вызывает для выполнения реальной работы.

Поскольку открытый метод не помечен как асинхронный, любые исключения, которые он генерирует, будут передаваться непосредственно вызывающей стороне. Но о любых сбоях, возникающих после выполнения работы в локальном методе, будет сообщено через задачу.

Листинг 17.21. Проверка аргументов для асинхронных методов

```
public static Task<string> FindLongestLineAsync(string url)
{
    if (url == null)
    {
        throw new ArgumentNullException("url");
    }
    return FindLongestLineCore(url);

    static async Task<string> FindLongestLineCore(string url)
    {
        ...
    }
}
```

Я решил переслать аргумент `url` локальному методу. Это было необязательно, потому что локальный метод может получить доступ к переменным содержащего его метода. Но на основе этого компилятор создает тип, который будет содержать локальные ресурсы для их совместного использования между методами. Там, где это возможно, он будет делать его значимым типом, передавая по ссылке внутреннему типу, но в тех случаях, когда область видимости внутреннего метода может пережить внешний метод, это невозможно. А поскольку локальный метод здесь асинхронный, то он, скорее всего, будет продолжать работать еще долго после того, как стековый фрейм внешнего метода перестанет существовать, что заставит компилятор создать ссылочный тип только лишь для хранения этого аргумента `url`. Передавая аргумент, мы этого избегаем (и я пометил метод как `static`, чтобы подтвердить свое намерение, — это означает, что компилятор выдаст ошибку, если я непреднамеренно использую в локальном методе что-либо из внешнего). Компилятору, конечно, по-прежнему придется создать объект, чтобы хранить локальные переменные во внутреннем методе во время асинхронного выполнения, но по крайней мере мы избежали создания большего количества объектов, чем было необходимо.

Единичные и множественные исключения

Как показано в главе 16, TPL определяет модель для сообщения о множественных ошибках — свойство `Exception` задачи возвращает `AggregateException`. Даже если имеется только один сбой, вы все равно должны извлечь его из `AggregateException`. Но если вы используете ключевое слово `await`, это бу-

дет сделано за вас — как вы видели в листинге 17.19, оно извлекает первое исключение в `InnerExceptions` и выбрасывает его еще раз.

Такое удобно, когда операция может вызвать только один сбой — это избавляет от необходимости писать дополнительный код для обработки составного исключения, а затем извлекать содержимое. (Если вы используете задачу, возвращаемую асинхронным методом, она никогда не будет содержать более одного исключения.) Тем не менее это ведет к проблеме, если вы работаете с составными задачами, в которых могут одновременно произойти несколько сбоев. Например, `Task.WhenAll` принимает набор задач и возвращает одну, которая завершается только после завершения всех составляющих ее задач. Если какие-то из них завершатся с ошибкой, вы получите исключение `AggregateException`, содержащее ряд ошибок. Если с такой операцией вы используете `await`, то вам вернется только первое из этих исключений.

Обычные механизмы TPL — метод `Wait` или свойство `Result` — предоставляют полный набор ошибок (создавая исключение `AggregateException` вместо его первого внутреннего исключения), но блокируют поток, если задача еще не завершена. Но что, если вам нужна эффективная асинхронная операция `await`, которая использует потоки только тогда, когда у них есть работа, но вам по-прежнему нужно видеть все ошибки? Листинг 17.22 показывает один из подходов.

Листинг 17.22. Ожидание без вызова исключения и `Wait`

```
static async Task CatchAll(Task[] ts)
{
    try
    {
        var t = Task.WhenAll(ts);
        await t.ContinueWith(
            x => {},
            TaskContinuationOptions.ExecuteSynchronously);
        t.Wait();
    }
    catch (AggregateException all)
    {
        Console.WriteLine(all);
    }
}
```

Пример использует `await`, чтобы воспользоваться естественными преимуществами асинхронных методов C#, но вместо вызова `await` для самой

составной задачи он устанавливает продолжение. Продолжение может успешно завершиться после завершения предшественника независимо от того, насколько успешно тот завершился. Тело этого продолжения пустое, так что ошибки в нем быть не может, а это значит, что `await` не выбросит здесь исключения. `Wait` вызовет `AggregateException`, если что-то пойдет не так, что позволит блоку `catch` увидеть все исключения. И, поскольку мы вызываем `Wait` только после завершения `await`, мы знаем, что задача уже завершена, поэтому вызов не будет блокироваться.

Единственным недостатком этого является то, что в итоге, для того чтобы мы имели возможность ожидания без исключений, мы настраиваем целую дополнительную задачу. Я настроил продолжение так, чтобы оно выполнялось синхронно, и это позволяет избежать планирования второй части работы через пул потоков, но здесь все еще наблюдается довольно заметная трата ресурсов. Более сложный, но более эффективный подход заключается в использовании `await` обычным способом, но с написанием обработчика исключений, который проверяет наличие других исключений, как показано в листинге 17.23.

Листинг 17.23. Поиск дополнительных исключений

```
static async Task CatchAll(Task[] ts)
{
    Task t = null;
    try
    {
        t = Task.WhenAll(ts);
        await t;
    }
    catch (Exception first)
    {
        Console.WriteLine(first);

        if (t != null && t.Exception.InnerException.Count > 1)
        {
            Console.WriteLine("I've found some more:");
            Console.WriteLine(t.Exception);
        }
    }
}
```

Это позволяет избежать создания дополнительной задачи, но недостатком является то, что обработка исключений выглядит немного странно.

Параллельные операции и пропущенные исключения

Самый простой способ использования `await` — это выполнять одно действие за другим, как если бы вы работали с синхронным кодом. Хотя может показаться, что строго последовательное выполнение работы не полностью использует потенциал асинхронного кода, оно намного эффективнее использует доступные потоки, чем синхронный эквивалент. Кроме этого, оно также хорошо работает в коде пользовательского интерфейса на стороне клиента, давая потоку пользовательского интерфейса свободно реагировать на ввод, даже когда идет работа. Но можно зайти еще дальше.

Можно запустить несколько частей одновременно — вызвать асинхронный API и вместо немедленного использования `await` сохранить результат в переменной, а затем начать выполнять другую часть работы, прежде ожидания обоих. Хотя этот вполне рабочий метод может сократить общее время выполнения ваших операций, неосторожный программист может попасть в ловушку, показанную в листинге 17.24.

Пример одновременно извлекает содержимое из двух URL. Запустив обе части работы, он использует два выражения `await` для получения результатов каждой из них и для отображения длины результирующих строк. Если операции завершатся успешно, это сработает, но этот код плохо обрабатывает ошибки. Если первая операция завершится ошибкой, код никогда не доберется до выполнения второго `await`. Это означает, что если вторая операция, в свою очередь, завершится неудачей, никто не узнает об исключении, которое она выдает. В итоге TPL обнаружит, что исключение прошло незамеченным, что приведет к возникновению события `UnobservedTaskException`. (В главе 16 мы обсуждали обработку пропущенных исключений в TPL.) Проблема в том, что это происходит очень редко — требуется, чтобы обе операции завершились ошибкой в короткий промежуток времени, поэтому при тестировании это очень легко упустить.

Листинг 17.24. Как не нужно запускать несколько параллельных операций

```
static async Task GetSeveral(IHttpClientFactory cf)
{
    using (HttpClient w = cf.CreateClient())
    {
        w.MaxResponseContentBufferSize = 2_000_000;

        Task<string> g1 = w.GetStringAsync("https://endjin.com/");
        Task<string> g2 = w.GetStringAsync("https://oreilly.com");
```

```
// ПЛОХО!
Console.WriteLine((await g1).Length);
Console.WriteLine((await g2).Length);
}
}
```

Чтобы не попасть в ловушку, тщательно обрабатывайте исключения — например, вы можете перехватывать любые исключения, возникающие в первом `await`, прежде чем приступать к выполнению второго. В качестве альтернативы можно использовать `Task.WhenAll` для ожидания всех задач как одной операции — в случае ошибки это приведет к появлению сбойной задачи вкупе с `AggregateException`, что позволит увидеть все ошибки. Конечно, как вы видели в предыдущем разделе, со множеством сбоев такого рода иметь дело крайне неудобно, если используется `await`. Но если вы хотите запустить несколько асинхронных операций и одновременно выполнять их все на лету, потребуется более сложный код для координации результатов, нежели при последовательном выполнении работы. Несмотря на это, ключевые слова `await` и `async` по-прежнему способны значительно облегчить жизнь.

Итог

Асинхронные операции не блокируют поток, из которого они вызываются, что делает их более эффективными, чем синхронные API, а это особенно важно на сильно загруженных машинах. Это также делает их пригодными для использования на стороне клиента, поскольку они позволяют выполнять длительную работу, не вызывая перезапуска интерфейса пользователя. Без языковой поддержки асинхронные операции может оказаться сложно правильно использовать, особенно при обработке ошибок в нескольких связанных операциях. Ключевое слово `await` в C# позволяет создавать асинхронный код, который выглядит как обычный синхронный. Все становится немного сложнее, если нужно, чтобы один метод управлял несколькими параллельными операциями. Но даже если вы напишите асинхронный метод, который выполняет задачи строго по порядку, вы получите преимущества более эффективного использования потоков в серверном приложении. Оно сможет одновременно поддерживать большее количество пользователей, поскольку каждая отдельная операция использует меньше ресурсов, а на стороне клиента вы получите преимущество в виде более отзывчивого пользовательского интерфейса.

Методы, которые используют `await`, должны быть помечены ключевым словом `async` и обычно должны возвращать `Task`, `Task<T>`, `ValueTask` или `ValueTask<T>`. (C# допускает возвращаемый тип `void`, но его следует использовать только тогда, когда у вас не осталось выбора.) Компилятор сделает так, что эта задача успешно завершится после того, как ваш метод вернется, или завершится с ошибкой, если ваш метод потерпел неудачу в любой момент своего выполнения. Поскольку `await` может использовать любую `Task` или `Task<T>`, это позволяет легко разделить асинхронную логику между несколькими методами, так как высокоуровневый метод может ожидать низкоуровневый асинхронный метод. Как правило, работа в конечном итоге завершается выполнением некого основанного на задачах API, но не обязательно, потому что `await` требует лишь выполнения определенного шаблона, — это ключевое слово примет любое выражение, для которого вы можете вызвать метод `GetAwaiter`, чтобы получить подходящий тип.

Эффективная работа с памятью

Как описано в главе 7, CLR осуществляет автоматическое управление памятью благодаря сборщику мусора (GC). Это обходится довольно дорого: если процессор тратит время на сборку мусора, это мешает ему работать более продуктивно. На ноутбуках и смартфонах сборщик мусора расходует заряд батареи. При облачных вычислениях, где вы платите за использованное процессорное время, дополнительная работа для процессора означает увеличение затрат. Более того, на компьютере со многими ядрами слишком большой объем работы сборщика мусора способен значительно снизить пропускную способность, поскольку некоторые ядра могут оказаться заблокированными в ожидании завершения работы сборщика.

Во многих случаях такое воздействие будет достаточно небольшим, чтобы не вызывать видимых проблем. Но, когда определенные типы программ работают под большой нагрузкой, затраты на сборку мусора могут преобладать в общем времени выполнения. В частности, если вы пишете код, который выполняет относительно простую, но повторяющуюся обработку, расходы на сборку мусора могут оказать существенное влияние на пропускную способность.

Поскольку команда Microsoft ASP.NET Core в свое время работала над повышением производительности своей платформы веб-сервера, в ранних версиях они часто сталкивались с жесткими ограничениями из-за расходов на сборку мусора. Чтобы позволить приложениям .NET их преодолеть, в C# 7.2 были добавлены различные функции, которые способны значительно сократить количество выделений памяти. Меньшее выделений памяти означает меньшее количество блоков памяти для восстановления при сборке мусора, так что это улучшение приводит к непосредственному снижению издержек на сборку мусора. Широко эти функции стали использоваться в ASP.NET Core 3.0. Эта версия обладает улучшенной производительностью во многих аспектах, но в случае простейшего теста производительности, известного как «простой текст» (часть набора веб-тестов производитель-

ности TechEmpower), этот выпуск повышает скорость обработки запросов более чем на 25%.

В некоторых специализированных сценариях скачок может быть еще более существенным. В 2019 году я работал над проектом, который обрабатывал диагностическую информацию от сетевого оборудования широкополосного провайдера (в виде пакетов RADIUS). Применение описанных в этой главе методов повысило скорость, с которой одно ядро ЦП в нашей системе могло обрабатывать сообщения, примерно с 300 тыс./с до примерно 7 млн/с.

Конечно, за это пришлось заплатить: эффективные в плане сборки мусора методы значительно усложняют код. И эта цена не всегда будет оправдана — хотя ASP.NET Core 3.0 по сравнению с предыдущей версией выглядит выигрышнее во всех тестах, лишь самый простой из них показывает улучшение на 25%. На практике улучшение будет зависеть от характера вашей рабочей нагрузки, и применение этих методов в некоторых типах приложений может не дать ощутимых улучшений. Поэтому, прежде чем думать об их использовании, необходимо воспользоваться инструментами мониторинга производительности и выяснить, сколько времени ваш код тратит на сборку мусора. Если это всего лишь несколько процентов, то улучшения на порядок ждать не стоит. Но если тестирование показывает, что есть пространство для заметных улучшений, следующим шагом будет вопрос, способны ли в этом помочь методы из данной главы. Итак, давайте начнем с изучения того, как именно эти новые методы могут помочь вам снизить расходы на сборку мусора.

(Не) копируйте это

Способ уменьшить расходы на сборку мусора состоит в том, чтобы выделять меньше памяти в куче. И самый важный метод этой минимизации — избегать копирования данных. Например, посмотрим на URL <http://example.com/books/1323?edition=6&format=pdf>. В нем есть несколько интересующих нас элементов: протокол (`http`), имя хоста (`example.com`) и строка запроса. Последняя имеет собственную структуру: это последовательность пар имя/значение. Очевидный способ работы с URL-адресом в .NET — это использовать тип `System.Uri`, как это показано в листинге 18.1.

Это удобно, но, получая значения данных четырех свойств, мы заставили `Uri` создать четыре строковых объекта в дополнение к исходному. Вполне можно представить умную реализацию `Uri`, которая распознает определен-

ные стандартные значения `Scheme`, такие как `http`, и всегда возвращает для них один и тот же экземпляр строки вместо выделения новых, но для всех остальных частей, скорее всего, придется выделять в куче новые строки.

Листинг 18.1. Деконструкция URL

```
var uri = new Uri("http://example.com/books/1323?edition=6&format=pdf");
Console.WriteLine(uri.Scheme);
Console.WriteLine(uri.Host);
Console.WriteLine(uri.AbsolutePath);
Console.WriteLine(uri.Query);
```

Код выдает следующие строки:

```
http
example.com
/books/1323
edition=6&format=pdf
```

Но есть и другой путь. Вместо того чтобы создавать новые строковые объекты для каждого раздела, можно воспользоваться тем, что вся нужная информация уже имеется в строке, содержащей сам URL. Не нужно копировать каждый раздел в новую строку, когда вместо этого мы можем просто следить за положением и длиной соответствующих разделов в строке. Вместо создания одной строки для каждого раздела нам понадобятся всего два числа. И, поскольку для чисел мы можем использовать значимые типы (например, `int` или `long` для очень длинных строк), нам не потребуются дополнительные объекты в куче, кроме той одной строки с полным URL-адресом. Например, формат (`http`) находится в позиции 0 и имеет длину 4. На рис. 18.1 показан каждый из элементов согласно их смещению и положению в строке.

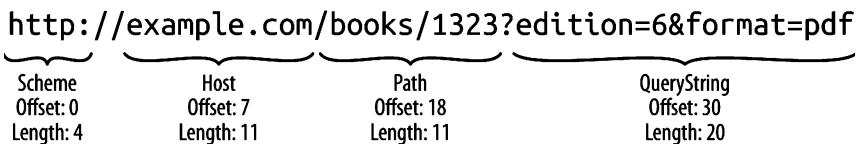


Рис. 18.1. Подстроки URL

Это работает, но уже мы можем видеть первую проблему этого метода: он несколько неуклюж. Вместо представления, скажем, `Host` в виде удобного строкового объекта, который легко понять и проверить в отладчике, у нас есть пара чисел, и мы как разработчики теперь должны помнить, на какую строку они указывают. Конечно, это не высшая математика, но понять наш код теперь

сложнее, а допустить ошибку проще. Но есть и преимущество: вместо пяти строк (исходный URL и четыре свойства) у нас теперь только одна. И, если вам нужно обрабатывать миллионы событий ежесекундно, усилия могут окупиться.

Очевидно, эта техника сработает и в менее масштабных случаях. Смещение и позиция (25, 4) в данном URL указывают на текст 1323. Возможно, нам понадобится преобразовать его в `int`. Но на этом этапе мы сталкиваемся со второй проблемой, связанной с этим стилем работы: он недостаточно широко поддерживается в библиотеках .NET. Обычный способ преобразования текста в `int` — это использование статических методов `Parse` или `TryParse` типа `int`. К сожалению, у них нет перегрузок, которые принимают позицию или смещение в `string`. Они требуют строку, содержащую только число, которое и будет проанализировано. Это означает, что в итоге вам придется написать код, подобный показанному в листинге 18.2.

Листинг 18.2. Перечеркиваем все преимущества использованием `Substring`

```
string uriString = "http://example.com/books/1323?edition=6&format=pdf";
int id = int.Parse(uriString.Substring(25, 4));
```

Это сработает, но, использовав `Substring` для перехода от нашего представления (смещение, длина) обратно к типу `string`, который требуется для `int.Parse`, мы выделили новую строку. Но смысл заключался в том, чтобы сократить выделения памяти, так что это совсем не похоже на шаг в нужном направлении. Одним из решений может стать пересмотр компанией Microsoft поверхности .NET API и добавление перегрузок, которые принимают параметры смещения и длины для ситуации, когда нам требуется работать с чем-то в середине чего-то еще (или подстрокой, как в этом примере, или, возможно, поддиапазоном массива). На самом деле примеры этого уже есть: `Stream API` для работы с потоками байтов содержит различные методы, которые принимают массив `byte[]`, а также аргументы `offset` и `length` для точного указания, с какой частью массива вы хотите работать.

Но есть и другая проблема, связанная с этой техникой: она негибкая в отношении типа контейнера, в котором находятся данные. Microsoft может добавить перегрузку к `int.Parse`, которая принимает `string`, смещение и длину, но она сможет анализировать данные только внутри `string`. Но что, если данные окажутся в `char[]`? В этом случае придется сначала преобразовать его в строку, в результате чего мы снова вернемся к дополнительным выделениям памяти. В качестве альтернативы каждому API, которому требуется поддерживать такой подход, потребуется несколько перегрузок

для поддержки всех контейнеров, которые только могут понадобиться, и каждый из них может потребовать различной реализации одного и того же базового метода.

А что, если ваши данные в настоящее время находятся в памяти, которая расположена не в куче CLR? Это крайне важный вопрос, когда речь идет о производительности серверов, которые принимают запросы по сети (например, веб-сервер). Иногда невозможно организовать размещение данных, полученных сетевой картой, непосредственно в памяти в куче .NET. Кроме того, некоторые формы межпроцессного взаимодействия задействуют организацию операционной системой отображения определенной области памяти в адресные пространства двух разных процессов. Куча .NET является локальной для процесса и не способна использовать такую память.

C# всегда поддерживал использование внешней памяти через небезопасный код, позволяющий работать с необработанными неуправляемыми указателями, поведение которых аналогично указателям в языках C и C++. Тем не менее с ними тоже есть несколько проблем. Во-первых, они бы дополнили список перегрузок, которые все должны поддерживать в окружении, где может потребоваться анализировать данные на месте. Во-вторых, код, использующий указатели, не пройдет проверку на соответствие правилам безопасности типов .NET. Это означает, что становятся возможными определенные виды ошибок программирования, которые обычно в C# не встретишь. Это также может означать, что коду не будет позволено запускаться в определенных сценариях, поскольку утрата безопасности типов позволит небезопасному коду обходить некоторые ограничения безопасности.

Подводя итог, скажу, что в .NET всегда можно было сократить выделение и копирование, работая со смещениями и длинами, а также либо со ссылкой на содержащую строку или массив, либо с неуправляемым указателем на память. Но вместе с тем всегда было много и возможностей для значительного улучшения по следующим фронтам:

- Удобство.
- Широкая поддержка в API .NET.
- Единообразная, безопасная работа со:
 - строками;
 - массивами;
 - неуправляемой памятью.

Но, начиная с C# 7.2, нам стал доступен тип, который затрагивает все три пункта: `Span<T>`. (Дополнительную информацию о том, как функции, описанные в этой главе, относятся к языку C# и версиям среды выполнения .NET, см. во врезке «Поддержка различных языков и версий среды выполнения».)

ПОДДЕРЖКА РАЗЛИЧНЫХ ЯЗЫКОВ И ВЕРСИЙ СРЕДЫ ВЫПОЛНЕНИЯ

Вас могли удивить мои слова о том, что в конкретной версии C# (7.2) появился новый тип. Вообще, новые типы определяются в библиотеках, поэтому они не привязаны к какой-либо конкретной версии C#, и на первый взгляд `Span<T>` выглядит просто еще одним типом. Он входит в состав базовых библиотек, которые поставляются с .NET Core, начиная с версии 2.1, а кроме того, имеется пакет NuGet, `System.Memory`, позволяющий использовать его в .NET Framework. Он также доступен для любой библиотеки, ориентированной на .NET Standard 2.1.

Но хотя `Span<T>` — это просто еще один тип, ему требуется C# 7.2 или новее, потому что определен он как `ref struct`. Более старые версии C# не поддерживают `ref struct` и поэтому не могут использовать `Span<T>`.

Помните, что эффективность методов, описанных в этой главе, зависит от версии .NET, которую вы используете. Хотя пакет NuGet `System.Memory` позволяет использовать обсуждаемые в этой главе типы, в программах, работающих на .NET Framework, вы получаете реализацию, немного отличающуюся от той, которая ждет вас при запуске точно такого же кода, на .NET Core 2.1 или более поздней версии. С этой версии .NET Core распознает `Span<T>` и родственные типы, а также обеспечивает специальные оптимизации, что очень важно для высокой производительности функций, описанных в этой главе.

В последней на момент написания версии .NET Framework (4.8) отсутствуют оптимизации `Span<T>`, и Microsoft уже не планирует добавлять их в будущих версиях, потому что на смену .NET Framework приходит .NET Core. Код, использующий эти техники, корректно работает в .NET Framework, но если вы хотите раскрыть все преимущества в производительности, вам придется работать в .NET Core.

Представление последовательных элементов с помощью `Span<T>`

Значимый тип `System.Span<T>` представляет собой последовательность элементов типа T, хранящуюся в памяти непрерывно. Эти элементы могут находиться внутри массива, строки, управляемого блока памяти в кадре

стека или в неуправляемой памяти. Давайте посмотрим, как Span<T> отвечает каждому из требований, перечисленных в предыдущем разделе.

Span<T> инкапсулирует три вещи: указатель или ссылку на содержащую память (например, string или массив), положение данных в этой памяти и длину. Чтобы получить доступ к содержимому подмножества, вы используете его аналогично массиву, как показано в листинге 18.3. Это делает данный тип гораздо более удобным в использовании, нежели специальные методы, в которых вы определяете пару переменных типа int и должны помнить, что на что ссылается¹.

Листинг 18.3. Итерация по Span<int>

```
public static int SumSpan(ReadOnlySpan<int> span)
{
    int sum = 0;
    for (int i = 0; i < span.Length; ++i)
    {
        sum += span[i];
    }
    return sum;
}
```

Поскольку Span<T> знает свою длину, его индексатор проверяет нахождение индекса в диапазоне, как это делает и встроенный тип массива. И если вы работаете в .NET Core, по производительности использование этого типа будет похоже на использование встроенного массива. В том числе это включает в себя оптимизацию, которая обнаруживает определенные шаблоны цикла — например, CLR распознает приведенный выше код как цикл, который выполняет итерацию по всему содержимому. Это позволяет генерировать код, которому не нужно каждый проход цикла проверять, находится ли индекс в допустимом диапазоне. В некоторых случаях он даже может создать код, который для ускорения работы цикла использует векторно ориентированные инструкции, доступные в некоторых процессорах. (На .NET Framework Span<T> работает немного медленнее, чем массив,

¹ В зависимости от того, на какой версии .NET вы работаете, первые два элемента могут быть объединены. .NET Core не хранит указатель и смещение отдельно: вместо этого он просто указывает непосредственно на нужные данные. В версии Span<T> для платформы .NET Framework необходимо хранить указатель отдельно, чтобы обеспечить правильную обработку подмножеств во время сборки мусора, поскольку в CLR не предусмотрены те же модификации для поддержки подмножеств, что сделаны в .NET Core.

потому что его CLR не содержит оптимизаций, которые были добавлены в .NET Core для поддержки `Span<T>`.)

Вы, возможно, заметили, что метод в листинге 18.3 принимает `ReadOnlySpan<T>`. Это близкий родственник `Span<T>`, и через неявное преобразование можно передать `Span<T>` методу, который принимает `ReadOnlySpan<T>`. Форма только для чтения позволяет методу указать, что он будет лишь читать из подмножества, но не записывать в него. (Это подтверждается тем фактом, что индексатор формы только для чтения содержит только метод доступа `get` и не содержит `set`.)



Всякий раз, когда вы пишете метод, который работает с подмножеством и не подразумевает его изменение, следует использовать `ReadOnlySpan<T>`.

Существуют также неявные преобразования из различных поддерживаемых контейнеров в `Span<T>` (а также в `ReadOnlySpan<T>`). Например, листинг 18.4 передает в метод `SumSpan` массив.

Листинг 18.4. Передача `int[]` как `ReadOnlySpan<int>`

```
Console.WriteLine(SumSpan(new int[] { 1, 2, 3 }));
```

Конечно, мы отошли от темы и разместили в куче массив, так что данный конкретный пример перечеркивает весь смысл использования подмножеств, но если у вас уже есть массив, этот метод будет полезен. `Span<T>` работает также и с массивами, выделенными в стеке, как показано в листинге 18.5. (Ключевое слово `stackalloc` позволяет вам создать массив в памяти, выделенной в текущем кадре стека.)

Листинг 18.5. Передача выделенного в стеке массива как `ReadOnlySpan<int>`

```
Span<int> numbers = stackalloc int[] { 1, 2, 3 };
Console.WriteLine(SumSpan(numbers));
```

Обычно C# не дает вам использовать `stackalloc` вне кода, помеченного как `unsafe`. Это ключевое слово выделяет память в фрейме стека текущего метода и не создает реальный объект массива. (Массивы являются ссылочными типами, поэтому должны располагаться в управляемой куче. Выражение `stackalloc` создает тип указателя, потому что выделяет просто память без обычных заголовков объекта .NET. В нашем случае это будет `int*`. Вы

можете непосредственно использовать типы указателей в небезопасных блоках кода.) Однако компилятор делает исключение из этого правила, если вы присвойте указатель, созданный через `stackalloc`, непосредственно подмножеству. Это разрешено, потому что подмножества навязывают проверку границ, предотвращая ошибки выхода за диапазон, которые обычно и делают указатели небезопасными. Кроме того, тот факт, что `Span<T>`, и `ReadOnlySpan<T>` — это `ref struct`, обеспечивает то, что подмножество не переживет свой содержащий фрейм стека, и это гарантирует, что кадр стека, в котором выделена память, не исчезнет, пока на него все еще имеются ссылки. (Правила проверки безопасности типов .NET включают специальную обработку для подмножеств.)

Ранее я упоминал, что подмножества могут ссылаться как на строки, так и на массивы. Однако мы не можем передать `string` этому `SumSpan` по той простой причине, что для этого требуется подмножество с типом элемента `int`, тогда как `string` представляет собой последовательность значений `char`. `int` и `char` имеют разные размеры — они занимают по 4 и 2 байта соответственно. Хотя между ними возможно неявное преобразование (т. е. вы можете присвоить значение `char` переменной типа `int`, что даст вам значение Unicode для `char`), это не делает `ReadOnlySpan<char>` неявно совместимым с `ReadOnlySpan<int>`. Помните, что вся идея подмножеств состоит в том, что они обеспечивают представление блока данных без необходимости копировать или изменять эти данные; поскольку `int` и `char` имеют разные размеры, преобразование `char[]` в `int[]` удвоит его размер². Однако если бы нам потребовался метод, принимающий `ReadOnlySpan<char>`, можно было бы передать ему `string`, `char[]`, `stackalloc char[]` или неуправляемый указатель типа `char*` (поскольку представление в памяти определенного подмножества символов в каждом случае является одинаковым).

Мы рассмотрели два требования из предыдущего раздела: `Span<T>` использовать проще, чем произвольно сохранять смещение и длину, и он позволяет написать единый метод, способный работать с данными в массивах, строках,

² Тем не менее возможно выполнить этот вид преобразования явно — класс `MemoryMarshal` содержит методы, которые могут принимать подмножество одного типа и возвращать другое подмножество, которое является представлением той же базовой памяти, но интерпретируется как содержащее другой тип элемента. Но это вряд ли пригодится в нашем случае: преобразование `ReadOnlySpan<char>` в `ReadOnlySpan<int>` произведет подмножество с половиной элементов, где каждый `int` будет состоять из пары смежных значений `char`.

стеке или неуправляемой памяти. Осталось последнее: широкая поддержка в библиотеках классов .NET. Как показано в листинге 18.6, теперь он поддерживается в `int.Parse`, что позволяет нам разобраться с проблемой, показанной в листинге 18.2.

Листинг 18.6. Разбор целых чисел в строке с использованием `Span<char>`

```
string uriString = "http://example.com/books/1323?edition=6&format=pdf";
int id = int.Parse(uriString.AsSpan(25, 4));
```



Так как строки в .NET являются неизменяемыми, нельзя преобразовать строку в `Span<char>`. Вы можете преобразовать ее только в `ReadOnlySpan<char>`.

Новые перегрузки, такие как эта, которые принимают подмножество там, где ранее принималась только строка или массив, сейчас уже получили распространение. Однако следует помнить, что эта работа еще не завершена. `Span<T>` — это относительно новый тип (он был введен в 2018 году, тогда как .NET существует с 2002 года), поэтому неизбежно остается множество сторонних библиотек, которые его еще не поддерживают и, возможно, никогда не поддержат. Более того, не все собственные библиотеки классов Microsoft в .NET Core 2.1 (первой версии, поддерживающей `Span<T>`) имеют его поддержку. Тем не менее работа ведется и ситуация будет только улучшаться.

Полезные методы

В дополнение к похожему на массив индексатору и свойствам `Length` `Span<T>` содержит несколько полезных методов. Методы `Clear` и `Fill` дают возможность удобно инициализировать все элементы в подмножестве либо значением по умолчанию для типа элемента, либо конкретным значением. Очевидно, что они недоступны для `ReadOnlySpan<T>`.

Иногда возникают ситуации, когда есть подмножество, нужно передать его содержимое в метод, который требует массив. Очевидно, что в этом случае не удастся избежать выделения памяти, но, если это неизбежно, можно воспользоваться методом `ToArray`.

Подмножества (как обычные, так и только для чтения) также содержат метод `TryCopyTo`, который в качестве аргумента принимает подмножество (не только для чтения) с тем же типом элемента. Метод позволяет копировать

данные между подмножествами. Он предназначен для сценариев, в которых исходное и целевое подмножества ссылаются на пересекающиеся области в одном и том же контейнере.

Только стек

И `Span<T>`, и `ReadOnlySpan<T>` объявлены как `ref struct`. Это означает, что они не только являются значимыми типами, но значимыми типами, которые могут существовать только в стеке. Таким образом, вы не можете иметь поля с типами подмножества в классе или любой структуре, которая, в свою очередь, не является `ref struct`. Это также накладывает и некоторые более неожиданные ограничения. Например, вы не можете использовать подмножество в переменной в асинхронном методе. (Они хранят все свои переменные в виде полей скрытого типа, что позволяет им располагаться в куче, потому что асинхронные методы могут пережить исходный фрейм стека. Фактически эти методы могут даже переключаться на совершенно другой стек, поскольку асинхронные методы могут в конечном итоге выполняться в разных потоках.) По аналогичным причинам существуют ограничения на использование подмножеств в анонимных функциях и в методах итераторов. Ими можно пользоваться в локальных методах, даже объявить переменную `ref struct` во внешнем методе и использовать ее из вложенного, но с одним ограничением: нельзя создавать делегат, который ссылается на этот локальный метод, потому что это вынудит компилятор переместить общие переменные в объект, который располагается в куче. (Подробности см. в главе 9.)

Это ограничение необходимо, чтобы в .NET стала возможна комбинация производительности, подобной массиву, безопасности типов и гибкости для работы с несколькими различными контейнерами. Для ситуаций, в которых это ограничение на использование только стека представляет проблему, у нас имеется тип `Memory<T>`.

Представление последовательных элементов с помощью Memory<T>

Тип `Memory<T>` и его собрат, `ReadOnlyMemory<T>`, в основном представляют собой ту же концепцию, что и `Span<T>` с `ReadOnlySpan<T>`. Эти типы обеспечивают единообразный вид непрерывной последовательности элементов

типа `T`, которые могут находиться в массиве, неуправляемой памяти или, если тип элемента — `char`, в `string`. Но в отличие от подмножеств `Span`, они не являются типами `ref struct`, поэтому их можно использовать где угодно. Обратной стороной медали является то, что они не дают такой же высокой производительности, как подмножества. (Это также означает, что вы не можете создать `Memory<T>` со ссылкой на память, выделенную с помощью `stackalloc`.)

Возможно конвертировать `Memory<T>` в `Span<T>` и аналогично `ReadOnlyMemory<T>` в `ReadOnlySpan<T>`. Это делает данные типы полезными, когда вы хотите иметь что-то подобное подмножествам, но в контексте, где они не разрешены (например, в асинхронном методе).



Преобразование в подмножество имеет свою цену. Не такую большую, но значительно более высокую, чем затраты на доступ к отдельному элементу в подмножестве. (В частности, многие из оптимизаций, которые делают подмножества привлекательными, становятся эффективными только при повторном использовании одного и того же подмножества.) Так что, если вы собираетесь в цикле читать или записывать элементы в `Memory<T>`, стоит задуматься об однократном преобразовании в `Span<T>` вне цикла, вместо того чтобы делать это каждый раз. Если вы можете работать только с подмножествами, это стоит делать, поскольку они дают наилучшую производительность. (А если производительность вас не интересует, тогда эта глава вообще не для вас!)

ReadOnlySequence<T>

Все типы, которые мы рассмотрели в этой главе, представляют собой непрерывные блоки памяти. К сожалению, данные не всегда приходят к нам в максимально удобной форме. Например, на занятом сервере, который обрабатывает много одновременных запросов, сетевые сообщения для выполняемых запросов часто чередуются — если конкретный запрос достаточно велик и его лучше разделить на два сетевых пакета, вполне возможно, что после получения первого и до получения второго может поступить один или несколько пакетов для других, не связанных с этим запросом. Таким образом, к тому времени, когда мы приступим к обработке содержимого запроса, он может быть разделен в памяти на две части. Поскольку значения подмножества и памяти могут представлять только непрерывный диапазон,

зон элементов, .NET включает в себя еще один тип, `ReadOnlySequence`, для представления данных, которые концептуально представляют собой одну последовательность, разбитую на несколько диапазонов.



Соответствующего типа `Sequence<T>` не существует. В отличие от подмножеств и памяти, эта конкретная абстракция доступна лишь в форме только для чтения. При чтении вполне нормально иметь дело с фрагментированными данными, когда вы не контролируете, где они хранятся. Но если вы производите данные, вы, скорее всего, способны контролировать, куда именно они попадают.

Теперь, когда мы рассмотрели основные типы для работы с данными при минимизации количества выделений, давайте узнаем, как все они могут совместно работать в задачах обработки больших объемов данных. Чтобы координировать этот вид обработки, нам нужно рассмотреть еще одну функцию: конвейеры.

Обработка потоков данных с помощью конвейеров

Все, о чем мы говорим в этой главе, предназначено для обеспечения безопасной и эффективной обработки больших объемов данных. Все типы, которые мы видели до сих пор, представляют информацию, которая уже находится в памяти. Кроме этого, нам также нужно подумать о том, как эти данные изначально попадают в память. В предыдущем разделе я намекнул на то, что это может происходить несколько запутанно. Данные очень часто будут разбиваться на части, но совсем не так, чтобы было удобно для обрабатывающего их кода. Связано это с тем, что они, скорее всего, будут поступать либо из сети, либо с диска. Если мы хотим реализовать преимущества производительности, ставшие возможными благодаря `Span<T>` и связанным с ним типам, нам в первую очередь следует обратить пристальное внимание на задачу загрузки данных в память и способ, которым этот процесс получения данных взаимодействует с кодом, который их обрабатывает. Даже если вы собираетесь писать лишь код, который использует данные, — скажем, в получении данных и размещении их в памяти — вы полагаетесь на такую среду, как ASP.NET Core, — важно понять, как этот процесс работает.

Пакет NuGet `System.IO.Pipelines` определяет набор типов в пространстве имен с тем же именем, которые обеспечивают высокопроизводительную загрузку данных из источника, который имеет тенденцию разбивать данные на куски неудобного размера и передавать их в код, который хочет их тут же обрабатывать, используя подмножества. На рис. 18.2 показаны основные участники процесса на основе конвейера.

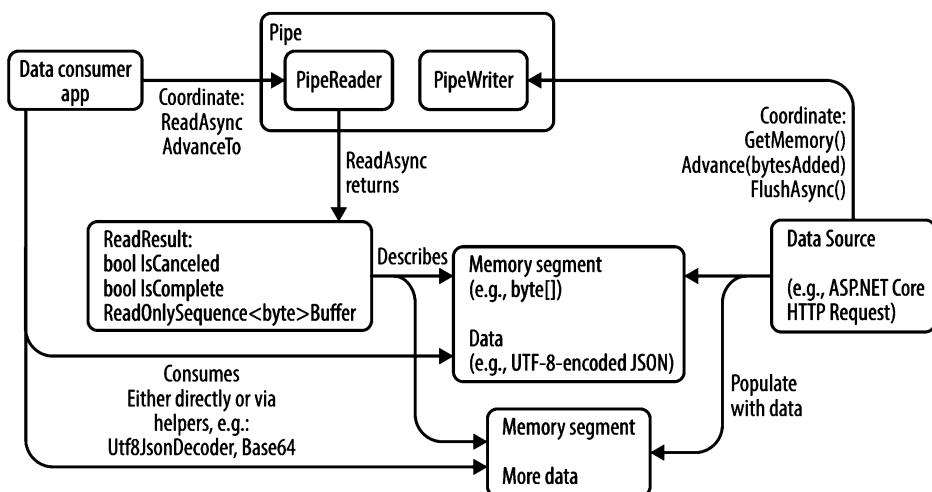


Рис. 18.2. Обзор конвейера

В основе всего лежит класс `Pipe`. Он содержит два свойства: `Writer` и `Reader`. Первое возвращает `PipeWriter`, и он используется кодом, который загружает данные в память. (Он часто не зависит от приложения. Скажем, в веб-приложении вы можете разрешить ASP.NET Core управлять записью от своего имени.) Тип свойства `Reader`, как и ожидалось, — `PipeReader`, и с ним то, скорее всего, и будет взаимодействовать ваш код.

Основной процесс чтения данных из конвейера заключается в следующем. Прежде всего вы вызываете `PipeReader.ReadAsync`. Этот метод возвращает задачу, потому что если данные еще не доступны, вам придется ждать, пока источник данных не предоставит данные для записи³. Как только данные станут доступны, задача предоставит объект `ReadResult`. Он, в свою очередь, предоставляет `ReadOnlySequence<T>`, где содержатся доступные данные в виде

³ Это `ValueTask<ReadResult>`, так как целью этого упражнения является минимизация числа выделений памяти. `ValueTask<T>` был описан в главе 16.

одного или нескольких значений `ReadOnlySpan<T>`. Количество подмножеств будет зависеть от фрагментации данных. Если все удобно расположено в одном месте в памяти, вы получите лишь один промежуток, но читающий код должен уметь справляться и с большим количеством. Следовательно, ваш код должен обрабатывать как можно больше доступных данных. После этого он вызывает `AdvanceTo` читателя, чтобы сообщить, сколько данных смог обработать ваш код. Затем, если свойство `ReadResult.IsComplete` имеет значение `false`, мы снова повторяем эти шаги, начиная с вызова `ReadAsync`.

Важной деталью является то, что нам разрешено сообщать `PipeReader`, что мы не в состоянии обработать все, что он нам предоставил. Обычно это происходит из-за того, что информация разбита на части и нам нужно увидеть следующую часть, прежде чем мы сможем полностью обработать текущий фрагмент. Например, сообщение JSON, достаточно большое, чтобы быть распределенным по нескольким сетевым пакетам, вероятно, будет разбито в самых неудобных местах. Например, первый фрагмент может выглядеть так:

```
{"property1": "value1", "prope
```

А второй так:

```
rty2": 42}
```

На практике части будут больше, но основная проблема налицо: куски, которые возвращает `PipeReader`, могут быть разделены прямо по середине важного свойства. С большинством API .NET вам никогда не придется столкнуться с подобным, потому что к моменту просмотра все было подчищено и собрано заново, но платой за это является выделение новых строк для хранения рекомбинированных результатов. Если вы хотите избежать этого распределения памяти, проблемы необходимо решить.

Есть несколько способов с ними разобраться. Один из них состоит в том, чтобы код чтения умел сохранять состояние для возможности остановиться и затем снова начать работу в любой точке последовательности. Так что при обработке этого JSON код может запомнить, что он находится на полпути чтения объекта и в середине обработки свойства, имя которого начинается с «`prope`». Но `PipeReader` предлагает другой способ. Код, обрабатывающий эти примеры, может сообщить своим вызовом `AdvanceTo`, что он получил все до первой запятой. Если это сделать, конвейер запомнит, что мы еще не закончили с этим первым блоком, и, когда следующий вызов `ReadAsync` завершится, `ReadOnlySequence<T>` в `ReadReult.Buffer` будет включать по меньшей мере два подмножества: первое будет указывать на тот же блок

памяти, что и в прошлый раз, но теперь его смещение будет установлено на то место, которое мы получили в прошлый раз, а именно на текст «probe» в конце первого блока. Ну а второе подмножество будет ссылаться на текст во втором куске.

Преимущество второго подхода состоит в том, что коду, обрабатывающему данные, не нужно слишком много запоминать между вызовами `ReadAsync`. Он знает, что, как только появится следующий фрагмент и это будет иметь смысл, он сможет вернуться и просмотреть ранее необработанные данные.

На практике с этим конкретным примером довольно легко разобраться, потому что в библиотеке классов имеется тип `Utf8JsonReader`, который способен помочь со всеми сложностями касательно границ фрагментов. Давайте посмотрим на реальный пример.

Обработка JSON в ASP.NET Core

Предположим, вы разрабатываете веб-сервис, который должен обрабатывать HTTP-запросы, содержащие JSON. Это довольно распространенный сценарий. В листинге 18.7 показан типовой способ сделать это в ASP.NET Core. Это достаточно простой пример, но в нем не используется ни один из механизмов экономного выделения памяти, обсуждаемых в этой главе, поэтому ASP.NET Core вынужденно выделяет по несколько объектов для каждого запроса.

Листинг 18.7. Обработка JSON в HTTP-запросах

```
[HttpPost]
[Route("/jobs/create")]
public void CreateJob([FromBody] JobDescription requestBody)
{
    switch (requestBody.JobCategory)
    {
        case "arduous":
            CreateArduousJob(requestBody.DepartmentId);
            break;

        case "tedious":
            CreateTediousJob(requestBody.DepartmentId);
            break;
    }
}
```

```
public class JobDescription
{
    public int DepartmentId { get; set; }
    public string JobCategory { get; set; }
}
```

Прежде чем мы будем думать, как его изменить, я быстро объясню, что происходит в этом примере, на случай, если вы не знакомы с ASP.NET Core. Метод `CreateJob` снабжен атрибутами, сообщающими ASP.NET Core, что он будет обрабатывать запросы HTTP POST, где путь в URL выглядит как `/jobs/create`. Атрибут `[FromBody]` в аргументе метода указывает, что, согласно нашим ожиданиям, тело запроса будет содержать данные в форме, описанной классом `JobDescription`. ASP.NET Core можно настроить на обработку различных форматов данных, но если вы используете значения по умолчанию, это будет JSON.

Поэтому в данном примере мы сообщаем ASP.NET Core, что для каждого POST-запроса к `/jobs/create` необходимо создать объект `JobDescription`, заполнив его `DepartmentId` и `JobCategory` из свойств с таким же именем в JSON в теле входящего запроса.

Другими словами, мы просим ASP.NET Core выделить два объекта — `JobDescription` и `string` — для каждого запроса, каждый из которых будет содержать копии информации, которая была в теле входящего запроса. (Другое свойство, `DepartmentId`, является типом `int`, и поскольку это значимый тип, он находится внутри объекта `JobDescription`.) И для большинства приложений это вполне подойдет — из-за пары выделений памяти в ходе обработки одного веб-запроса обычно волноваться не стоит. Однако в более реалистичных сценариях с более сложными запросами мы бы имели дело с гораздо большим количеством свойств, а если вам нужно обработать очень большой объем запросов, то копирование данных в `string` для каждого свойства способно дать дополнительную работу сборщику мусора, что приводит к проблемам с производительностью.

Листинг 18.8 показывает, как можно избежать этих выделений памяти, используя различные функции, описанные в предыдущих разделах этой главы. Это делает код намного более сложным и показывает, почему вы должны применять эти методы только в тех случаях, когда вам удалось выяснить, что издержки на сборку мусора достаточно высоки, чтобы дополнительные усилия на разработку оправдались.

Листинг 18.8. Обработка JSON без выделения ресурсов

```
private static readonly byte[] Utf8TextJobCategory =
    Encoding.UTF8.GetBytes("JobCategory");
private static readonly byte[] Utf8TextDepartmentId =
    Encoding.UTF8.GetBytes("DepartmentId");
private static readonly byte[] Utf8TextArduous =
    Encoding.UTF8.GetBytes("arduous");
private static readonly byte[] Utf8TextTedious =
    Encoding.UTF8.GetBytes("tedious");

[HttpPost]
[Route("/jobs/create")]
public async ValueTask CreateJobFrugalAsync()
{
    bool inDepartmentIdProperty = false;
    bool inJobCategoryProperty = false;
    int? departmentId = null;
    bool? isArduous = null;

    PipeReader reader = this.Request.BodyReader;
    JsonReaderState jsonState = default;
    while (true)
    {
        ReadResult result =
            await reader.ReadAsync().ConfigureAwait(false);
        jsonState = ProcessBuffer(
            result,
            jsonState,
            out SequencePosition position);

        if (departmentId.HasValue && isArduous.HasValue)
        {
            if (isArduous.Value)
            {
                CreateArduousJob(departmentId.Value);
            }
            else
            {
                CreateTediousJob(departmentId.Value);
            }
        }

        return;
    }

    reader.AdvanceTo(position);
```

```
if (result.IsCompleted)
{
    break;
}

JsonReaderState ProcessBuffer(
    in ReadResult result,
    in JsonReaderState jsonState,
    out SequencePosition position)
{
    // Это ref struct, которая не требует дополнительных сборок
    // мусора
    var r = new Utf8JsonReader(result.Buffer, result.IsCompleted,
        jsonState);

    while (r.Read())
    {
        if (inDepartmentIdProperty)
        {
            if (r.TokenType == JsonTokenType.Number)
            {
                if (r.TryGetInt32(out int v))
                {
                    departmentId = v;
                }
            }
        }
        else if (inJobCategoryProperty)
        {
            if (r.TokenType == JsonTokenType.String)
            {
                if (r.ValueSpan.SequenceEqual(Utf8TextArduous))
                {
                    isArduous = true;
                }
                else if (
                    r.ValueSpan.SequenceEqual(Utf8TextTedious))
                {
                    isArduous = false;
                }
            }
        }
    }

    inDepartmentIdProperty = false;
    inJobCategoryProperty = false;
}
```

```
        if (r.TokenType == JsonTokenType.PropertyName)
    {
        if (r.ValueSpan.SequenceEqual(Utf8TextJobCategory))
        {
            inJobCategoryProperty = true;
        }
        else if (r.ValueSpan.SequenceEqual(Utf8TextDepartmentId))
        {
            inDepartmentIdProperty = true;
        }
    }
}

position = r.Position;
return r.CurrentState;
}
}
```

Вместо определения аргумента с атрибутом [FromBody] данный метод работает непосредственно со свойством `this.Request.BodyReader`. (Внутри класса контроллера MVC ASP.NET Core метод `this.Request` возвращает объект, представляющий собой обрабатываемый запрос.) Тип этого свойства — `PipeReader`, принимающая сторона `Pipe`. ASP.NET Core создает конвейер и управляет стороной производства данных, передавая данные из входящих запросов в связанный `PipeWriter`.

Как следует из названия свойства, этот конкретный `PipeReader` позволяет нам читать содержимое тела HTTP-запроса. Считывая данные таким образом, мы даем ASP.NET Core возможность тут же представлять нам тело запроса: наш код сможет считывать данные непосредственно из места в памяти, куда они были помещены как только сетевая карта компьютера их получила. (Другими словами, никаких копий и дополнительных затрат на сборку мусора.)

Цикл `while` в `CreateJobFrugalAsync` работает так же, как и в любом коде, который читает данные из `PipeReader`: он вызывает `ReadAsync`, обрабатывает возвращаемые данные и вызывает `AdvanceTo`, чтобы `PipeReader` знал, какую часть этих данных удалось обработать. Затем мы проверяем свойство `IsComplete` объекта `ReadResult`, возвращенного `ReadAsync`, и, если оно равно `false`, мы повторяем все еще раз.

В листинге 18.8 для чтения данных используется тип `Utf8JsonReader`. Как следует из названия, он работает напрямую с текстом в кодировке UTF-8.

Одно это может дать значительный прирост производительности: сообщения JSON обычно отправляются в этой кодировке, но строки .NET используют UTF-16. Поэтому одной из задач, которые листинг 18.7 поставил перед ASP.NET, было преобразование всех строк из UTF-8 в UTF-16. С другой стороны, мы потеряли часть гибкости. Преимущество более простого и медленного подхода заключается в возможности адаптации ко входящим запросам в гораздо большем количестве форматов: если клиент решил отправить свой запрос в другом формате, отличном от UTF-8, например UTF-16 или UCS-32, или даже в кодировке, отличной от Unicode, такой как ISO-8859-1, то наш обработчик сможет справиться с любым из них, потому что ASP.NET Core способен выполнять преобразования строк за нас. Но поскольку листинг 18.8 работает с данными непосредственно в той форме, в которой их передал клиент, при этом используя тип, который понимает только UTF-8, мы потеряли эту гибкость в обмен на более высокую производительность.

`Utf8JsonReader` способен справиться с не самыми простыми проблемами: если входящий запрос из-за своего размера разбивается на несколько буферов в памяти (он слишком велик для размещения в одном сетевом пакете), `Utf8JsonReader` поможет разобраться с этим. При таком случайному разбиении он обработает все, что сможет, после чего возвращаемое через `CurrentState` значение `JsonReaderState` будет содержать позицию первого необработанного символа. Его мы передаем в `PipeReader.AdvanceTo`. Следующий вызов `PipeReader.ReadAsync` будет возвращаться только при наличии дополнительных данных, но его `ReadReultsult.Buffer` также будет включать и ранее не использованные данные.

Как и тип `ReadOnlySpan<T>`, который он использует при чтении данных, `Utf8JsonReader` — это тип `ref struct`, что означает, что он не может располагаться в куче. Это означает, что его нельзя использовать в асинхронном методе, потому что асинхронные методы хранят все свои локальные переменные в куче. Вот почему в этом примере есть отдельный метод `ProcessBuffer`. Внешний метод `CreateJobFrugalAsync` должен быть асинхронным, потому что потоковая природа типа `PipeReader` означает, что его метод `ReadAsync` требует от нас использования `await`. Но `Utf8JsonReader` нельзя использовать в асинхронном методе, поэтому мы вынуждены разделить нашу логику на два метода.

При использовании `Utf8JsonReader` наш код должен быть готов к приему содержимого в любом порядке поступления. Недопустимо писать код, который пытается прочитать свойства в удобном для нас порядке, потому

что порядок будет зависеть от того, что хранит эти свойства и их значения в памяти. (Если вы попытаетесь вернуться к базовым данным для получения определенных свойств по требованию, вы можете столкнуться с тем, что требуемое свойство было в более раннем фрагменте, который более не доступен.) Это сводит на нет всю суть минимизации выделений памяти. Если вы хотите избежать выделения ресурсов, ваш код должен быть достаточно гибким, чтобы обрабатывать свойства в любом порядке их появления.



При разделении обработки конвейера на внешний цикл асинхронного чтения и внутренний метод, который избегает асинхронности ради использования типов `ref struct`, может оказаться удобнее сделать внутренний метод локальным, как в листинге 18.8. Это позволяет получить доступ к переменным, объявленным во внешнем методе. Вам может быть интересно, не вызывает ли это скрытое дополнительное выделение памяти. Чтобы разрешить совместное использование переменных таким способом, компилятор генерирует тип, сохраняя общие переменные в полях этого типа, а не как обычные переменные на основе стека. В случае лямбда-выражений и других анонимных методов, этот тип действительно вызовет дополнительное выделение, потому что он должен будет пребывать в куче, чтобы иметь возможность пережить родительский метод. Но в локальных методах компилятор использует `struct` для хранения общих переменных, которые он передает по ссылке на внутренний метод, что позволяет избежать дополнительного выделения памяти. Это становится возможным, потому что компилятор способен определить, что все вызовы локального метода вернутся до того, как вернется внешний метод.

Поэтому код `ProcessBuffer` в листинге 18.8 просто просматривает каждый элемент JSON по мере его поступления и определяет, представляет ли тот интерес. Это означает, что при поиске определенных значений свойств мы должны отметить элемент `PropertyName`, а затем запомнить, что он был последним из увиденных нами. Тем самым мы будем знать, как обрабатывать следующий элемент `Number` или `String`, содержащий значение.

Одна неожиданная и странная особенность этого кода — это его способ поиска определенных строк. Ему необходимо распознавать интересующие его свойства (в данном случае `JobCategory` и `DepartmentId`), но мы не можем просто использовать обычное сравнение строк. Хотя можно получить имена свойств и строковые значения в виде строк .NET, это удаляет нас от основной цели использования `Utf8JsonReader`, так как если вы получаете

`string`, CLR придется выделить для нее место в куче и в конечном итоге осуществить сборку мусора. (В этом примере каждая допустимая входящая строка известна заранее. В некоторых сценариях для дальнейшей обработки вам понадобятся значения переданных пользователем строк, и в таких случаях придется смириться с затратами на выделение памяти для экземпляра `string`.) Поэтому вместо этого мы в итоге выполняем двоичные сравнения. Обратите внимание, что мы работаем только в кодировке UTF-8, а не в кодировке UTF-16, используемой строковым типом .NET. (Различные статические поля, такие как `Utf8TextJobCategory` и `Utf8TextDepartmentId`, являются байтовыми массивами, созданными с помощью `Encoding.UTF8` из пространства имен `System.Text`.) Это происходит потому, что весь этот код непосредственно работает с запросом в той форме, в которой он поступил через сеть, что позволяет избежать ненужного копирования.

Итог

API, которые разбивают данные на составляющие, могут быть очень удобны в использовании, но у этого удобства есть своя цена. Каждый раз, когда нужно, чтобы какой-то подэлемент был представлен в виде строки или дочернего объекта, нам приходится размещать еще один объект в управляемой куче. Совокупная стоимость этих выделений памяти (и соответствующих работ по восстановлению памяти после использования) может нанести ущерб в ряде чувствительных к производительности приложений. Затраты могут оказаться значительными в облачных приложениях или при обработке больших объемов данных, где вы можете платить за объем выполняемой вами обработки, — сокращение использования ЦП или памяти может оказать заметное влияние на стоимость.

Тип `Span<T>` и родственные ему типы, обсуждаемые в этой главе, позволяют работать с данными вне зависимости от места их расположения в памяти. Обычно для этого требуется более сложный код, но в случаях, когда отдача оправдывает лишнюю работу, эти функции позволяют C# решать целые классы задач, для которых раньше он был слишком медленным.

Об авторе

Иэн Гриффитс работает в компании endjin техническим специалистом. Живет в Хоуве, Англия. Его имя часто мелькает в рассылках для разработчиков и в группах новостей, где подписчики соревнуются в том, чтобы заставить Иэна написать наидлиннейший ответ на самый короткий вопрос. Иэн является соавтором «Windows Forms in a Nutshell», «Mastering Visual Studio .NET» и «Programming WPF».

Об обложке

На обложке — венценосный журавль (*Balearica regulorum*). Ареал обитания этой птицы простирается от Кении и Уганды на севере до восточной части Южной Африки. Они предпочитают жить на болотах и лугах.

Взрослые особи вырастают от 90 до 123 см и весят около 3,5 кг. Это великолепные птицы с серым телом и бледно-серой шеей, бело-золотыми крыльями, белой же лицевой частью (с красным пятнышком сверху), черной шапочкой, ярко-красным горловым мешочком и голубыми глазами. Венчает все это (поэтому птица так и называется) потрясающий хохолок из жестких золотистых перьев.

Венценосные журавли могут жить в дикой природе до 20 лет и проводят большую часть своего времени, выискивая в траве мелких животных, насекомых и семена. Они являются одним из двух видов журавлей, способных ночевать на деревьях, и этим искусством обязаны цепкому заднему пальцу, который позволяет им удерживаться на ветвях. Птицы откладывают до четырех яиц, и уже через несколько часов после вылупления птенцы могут следовать за своими родителями, поэтому корм семья добывает вместе.

Общительные и говорливые, венценосные журавли создают пары или семьи, которые иногда объединяются в стаи из более чем ста птиц. Как и другие журавли, они знамениты своим сложным брачным танцем, который включает в себя короткие взлеты вверх, взмахи крыльев и глубокие поклоны.

Несмотря на широкий ареал обитания, эти птицы в настоящее время находятся под угрозой исчезновения из-за сокращения мест обитания, браконьерства и использования пестицидов. Многие из животных на обложках O'Reilly находятся под угрозой исчезновения; все они важны для мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры из «Естественной истории Касселла» (1896).

Иэн Гриффитс

Программируем на C# 8.0. Разработка приложений

Перевел с английского Р. Чикин

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>К. Тульцева</i>
Литературный редактор	<i>М. Петруненко</i>
Обложка	<i>В. Мостипан</i>
Корректор	<i>М. Одинокова, Н. Петрова</i>
Верстка	<i>Е. Неволайнен</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2021. Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.05.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 76,110. Тираж 700. Заказ

C++ для профи

Джош Лоспинозо



C++ – популярный язык для создания ПО. В руках увлеченного программиста C++ становится прекрасным инструментом для создания лаконичного, эффективного и читаемого кода, которым можно гордиться. «C++ для профи» адресован программистам среднего и продвинутого уровней, вы проредетесь сквозь тернии к самому ядру C++. Часть 1 охватывает основы языка C++ от типов и функций до жизненного цикла объектов и выражений. В части 2 представлена стандартная библиотека C++ и библиотеки Boost. Вы узнаете о специальных вспомогательных классах, структурах данных и алгоритмах, а также о том, как управлять файловыми системами и создавать высокопроизводительные программы, которые обмениваются данными по сети.

[КУПИТЬ](#)

Изучаем Python: программирование игр, визуализация данных, веб-приложения

3-е изд.

Эрик Мэтис



купить

«Изучаем Python» — это самое популярное в мире руководство по языку Python. Вы сможете не только максимально быстро его освоить, но и научитесь писать программы, устранять ошибки и создавать работающие приложения. В первой части книги вы познакомитесь с основными концепциями программирования, такими как переменные, списки, классы и циклы, а простые упражнения приучат вас к шаблонам чистого кода. Вы узнаете, как делать программы интерактивными и как протестировать код, прежде чем добавлять в проект. Во второй части вы примените новые знания на практике и создадите три проекта: аркадную игру в стиле Space Invaders, визуализацию данных с удобными библиотеками Python и простое веб-приложение, которое можно быстро развернуть онлайн.

Работая с книгой, вы научитесь:

- Использовать мощные библиотеки и инструменты Python: Pygame, Matplotlib, Plotly и Django.
- Создавать 2D-игры разной сложности, которыми можно управлять с клавиатуры и мыши.
- Создавать интерактивную визуализацию данных.
- Разрабатывать, настраивать и развертывать веб-приложения.
- Разбираться с багами и ошибками.

Новое издание было тщательно переработано и отражает последние достижения в практиках программирования на Python.

Философия Java

4-е полное изд.

Б. Эккель



Впервые читатель может познакомиться с полной версией этого классического труда, который ранее на русском языке печатался в сокращении. Книга, выдержанная в оригинале не одно переиздание, за глубокое и поистине философское изложение тонкостей языка Java считается одним из лучших пособий для программистов. Чтобы по-настоящему понять язык Java, необходимо рассматривать его не просто как набор неких команд и операторов, а понять его «философию», подход к решению задач в сравнении с таковыми в других языках программирования. На этих страницах автор рассказывает об основных проблемах написания кода: в чем их природа и какой подход использует Java в их решении. Поэтому обсуждаемые в каждой главе черты языка неразрывно связаны с тем, как они используются для решения определенных задач.

КУПИТЬ

Head First. Изучаем Go

Джей Макгаврен



Go упрощает построение простых, надежных и эффективных программ. А эта книга сделает его доступным для обычных программистов. Основная задача Go — эффективная работа с сетевыми коммуникациями и многопроцессорной обработкой, но код на этом языке пишется и читается не сложнее чем на Python и JavaScript. Простые примеры позволяют познакомиться с языком в действии и сразу приступить к программированию на Go. Так что вы быстро освоите общепринятые правила и приемы, которые позволяют вам называть себя гофером.

[КУПИТЬ](#)