Язык программирования C# 9 и платформа .NET 5

ОСНОВНЫЕ ПРИНЦИПЫ И ПРАКТИКИ ПРОГРАММИРОВАНИЯ

10-Е ИЗДАНИЕ

Эндрю Троелсен, Филипп Джепикс

Диалектика
www.dialektika.com

Apress®
www.apress.com

Язык программирования C# 9 и платформа .NET 5:

ОСНОВНЫЕ ПРИНЦИПЫ И ПРАКТИКИ ПРОГРАММИРОВАНИЯ

10-е издание

1 том

Pro C# 9 with .NET 5

FOUNDATIONAL PRINCIPLES AND PRACTICES IN PROGRAMMING

Tenth Edition

Andrew Troelsen
Philip Japikse

Apress®

Язык программирования C# 9 и платформа .NET 5:

ОСНОВНЫЕ ПРИНЦИПЫ И ПРАКТИКИ ПРОГРАММИРОВАНИЯ

10-е издание

1 том

Эндрю Троелсен
Филипп Джепикс

Київ
Комп'ютерне видавництво
"ДІАЛЕКТИКА"
2022

УДК 004.432
Т70

Перевод с английского и редакция *Ю.Н. Артеменко*

Троелсен, Э., **Джепикс**, Ф.
Т70 Язык программирования C# 9 и платформа .NET 5: основные принципы и практики программирования, том 1, 10-е изд./Эндрю Троелсен, Филипп Джепикс; пер. с англ. Ю.Н. Артеменко. — Киев. : "Диалектика", 2022. — 770 с. : ил. — Парал. тит. англ.
ISBN 978-617-7987-81-8 (укр., том 1)
ISBN 978-617-7987-80-1 (укр., многотом.)
ISBN 978-1-4842-6938-1 (англ.)

В 10-м издании книги описаны новейшие возможности языка C# 9 и .NET 5 вместе с подробным "закулисным" обсуждением, призванным расширить навыки критического мышления разработчиков, когда речь идет об их ремесле. Книга охватывает ASP.NET Core, Entity Framework Core и многое другое наряду с последними обновлениями унифицированной платформы .NET, начиная с улучшений показателей производительности настольных приложений Windows в .NET 5 и обновления инструментария XAML и заканчивая расширенным рассмотрением файлов данных и способов обработки данных. Все примеры кода были переписаны с учетом возможностей последнего выпуска C# 9.

УДК 004.432

Все права защищены.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Apress Media, LLC.

Copyright © 2021 by Andrew Troelsen, Phillip Japikse.

All rights reserved.

Authorized translation from the *Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming* (ISBN 978-1-4842-6938-1), published by Apress Media, LLC.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher.

ISBN 978-617-7987-81-8 (укр., том 1)
ISBN 978-617-7987-80-1 (укр., многотом.)
ISBN 978-1-4842-6938-1 (англ.)

© "Диалектика", перевод, 2022

© 2021 by Andrew Troelsen, Phillip Japikse

ОГЛАВЛЕНИЕ

СОДЕРЖАНИЕ

Моей семье, Эми, Коннеру, Логану и Скайлер.
Спасибо за поддержку и терпение с вашей стороны.

Также моему отцу (Кору). Ты отец, муж, фантазер
и вообще верх совершенства для меня.

Филипп

Об авторах

Эндрю Троелсен обладает более чем 20-летним опытом работы в индустрии программного обеспечения (ПО). На протяжении этого времени он выступал в качестве разработчика, преподавателя, автора, публичного докладчика и теперь является руководителем команды и ведущим инженером в компании Thomson Reuters. Он был автором многочисленных книг, посвященных миру Microsoft, в которых раскрывалась разработка для COM на языке C++ с помощью ATL, COM и взаимодействия с .NET, а также разработка на языках Visual Basic и C# с использованием платформы .NET. Эндрю Троелсен получил степень магистра в области разработки ПО (MSSE) в Университете Сейнт Томас и трудится над получением второй степени магистра по математической лингвистике (CLMS) в Вашингтонском университете.

Филипп Джепикс — международный докладчик, обладатель званий Microsoft MVP, ASPInsider, профессиональный преподаватель по Scrum, а также активный участник сообщества разработчиков. Филипп имел дело еще с самыми первыми бета-версиями платформы .NET, разрабатывая ПО свыше 35 лет, и с 2005 года интенсивно вовлечен в сообщество гибкой разработки. Он является ведущим руководителем группы пользователей .NET и "круглого стола" по архитектуре ПО в Цинциннати, основанных на конференции CincyDeliver, а также волонтером Национального лыжного патруля. В настоящее время Филипп работает главным инженером и главным архитектором в Pintas & Mullins. Он любит изучать новые технологии и постоянно стремится совершенствовать свои навыки. Вы можете следить за деятельностью Филиппа в его блоге (`skimedic.com`) или в Твиттере (`@skimedic`).

О технических рецензентах

Аарон Стенли Кинг — опытный разработчик, который трудился в сфере цифрового маркетинга и помогал строить платформы SaaS на протяжении более 20 лет. Он считает программирование не только своей профессией, но и хобби, ставшим частью жизни. Аарон полагает, что компьютеры и технологии помогают вести ему более полноценную жизнь и максимально эффективно использовать свое время. Ему нравится рассказывать в группах пользователей и на конференциях о своем опыте и умениях. Аарон также вносит вклад в технологию открытого исходного кода. Он ведет блог на `www.aaronstanleyking.com`, и его можно отслеживать в Твиттере (`@trendoid`).

Брендон Робертс десять лет проработал помощником шерифа, и это привело к тому, что он стал детективом, занимающимся компьютерно-технической экспертизой. В данной роли он обнаружил, что ему нравится работать в сфере технологий. Получив травму при исполнении служебных обязанностей, Брендон решил взять инвалидность и заняться изучением разработки ПО. Он уже пять лет как профессиональный разработчик.

Эрик Смит — консультант в компании Strategic Data Systems (Шаронвилл, Огайо), работающий в команде проектов .NET. В 2017 году он окончил учебный курс по .NET от MAX Technical Training, а до того в 2014 году получил степень магистра по германистике в Университете Цинциннати. Эрик занимается разработкой ПО, начиная с середины 1990-х годов, и до сих пор любит писать код непосредственно для оборудования, когда появляется такая возможность. Помимо компьютеров большую часть времени он проводит за чтением, работой в своей механической мастерской и велоспортом на выносливость.

Благодарности

Я хочу поблагодарить издательство Apress и всю команду, вовлеченную в работу над данной книгой. Как я и ожидал в отношении всех книг, издаваемых в Apress, меня впечатлил тот уровень поддержки, который мы получали в процессе написания. Я благодарю вас, читатель, и надеюсь, что наша книга окажется полезной в вашей карьере, как было в моем случае. Наконец, я не сумел бы сделать эту работу без моей семьи и поддержки, которую получал от них. Без вашего понимания того, сколько времени занимает написание и вычитывание, мне никогда не удалось бы завершить работу! Люблю вас всех!

Филипп Джепикс

Введение

Авторы и читатели — одна команда

Авторам книг по технологиям приходится писать для очень требовательной группы людей (по вполне понятным причинам). Вам известно, что построение программных решений с применением любой платформы или языка исключительно сложно и специфично для отдела, компании, клиентской базы и поставленной задачи. Возможно, вы работаете в индустрии электронных публикаций, разрабатываете системы для правительства или местных органов власти либо сотрудничаете с NASA или какой-то военной отраслью. Вместе мы трудимся в нескольких отраслях, включая разработку обучающего ПО для детей (Oregon Trail/Amazon Trail), разнообразных производственных систем и проектов в медицинской и финансовой сферах. Написанный вами код на месте вашего трудоустройства почти на 100% будет иметь мало общего с кодом, который мы создавали на протяжении многих лет.

По указанной причине в книге мы намеренно решили избегать демонстрации примеров кода, свойственного какой-то конкретной отрасли или направлению программирования. Таким образом, мы объясняем язык C#, объектно-ориентированное программирование, .NET Runtime и библиотеки базовых классов .NET Core с использованием примеров, не привязанных к отрасли. Вместо того чтобы заставлять каждый пример наполнять сетку данными, подчитывать фонд заработной платы или выполнять другую задачу, специфичную для предметной области, мы придерживаемся темы, близкой каждому из нас: автомобили (с добавлением умеренного количества геометрических структур и систем расчета заработной платы для сотрудников). И вот тут наступает ваш черед.

Наша работа заключается в как можно лучшем объяснении языка программирования C# и основных аспектов платформы .NET 5. Мы также будем делать все возможное для того, чтобы снарядить вас инструментами и стратегиями, которые необходимы для продолжения обучения после завершения работы с данной книгой.

Ваша работа предусматривает усвоение этой информации и ее применение к решению своих задач программирования. Мы полностью отдаем себе отчет, что ваши проекты с высокой вероятностью не будут связаны с автомобилями и их дружественными именами, но именно в том и состоит суть прикладных знаний.

Мы уверены, что после освоения тем и концепций, представленных в настоящей книге, вы сможете успешно строить решения .NET 5, которые соответствуют вашей конкретной среде программирования.

Краткий обзор книги

Книга логически разделена на девять частей, каждая из которых содержит связанные друг с другом главы. Ниже приведено краткое содержание частей и глав.

Часть I. Язык программирования C# и платформа .NET 5

Эта часть книги предназначена для ознакомления с природой платформы .NET 5 и различными инструментами разработки, которые используются во время построения приложений .NET 5.

Глава 1. Введение в C# и .NET (Core) 5

Первая глава выступает в качестве основы для всего остального материала. Ее основная цель в том, чтобы представить вам набор строительных блоков .NET Core, таких как исполняющая среда .NET Runtime, общая система типов CTS, общеязыковая спецификация CLS и библиотеки базовых классов (BCL). Здесь вы впервые взглянете на язык программирования C#, пространства имен и формат сборок .NET 5.

Глава 2. Создание приложений на языке C#

Целью этой главы является введение в процесс компиляции файлов исходного кода C#. После установки .NET 5 SDK и исполняющей среды вы узнаете о совершенно бесплатном (но полнофункциональном) продукте Visual Studio Community, а также об исключительно популярном (и тоже бесплатном) продукте Visual Studio Code. Вы научитесь создавать, запускать и отлаживать приложения .NET 5 на языке C# с использованием Visual Studio и Visual Studio Code.

Часть II. Основы программирования на C#

Темы, представленные в этой части книги, очень важны, поскольку они связаны с разработкой ПО .NET 5 любого типа (например, веб-приложений, настольных приложений с графическим пользовательским интерфейсом, библиотек кода, служб и т.д.). Здесь вы узнаете о фундаментальных типах данных .NET 5, освоите манипулирование текстом и ознакомитесь с ролью модификаторов параметров C# (включая необязательные и именованные аргументы).

Глава 3. Главные конструкции программирования на C#: часть 1

В этой главе начинается формальное исследование языка программирования C#. Здесь вы узнаете о роли метода `Main()`, операторах верхнего уровня (нововведение в версии C# 9.0) , а также о многочисленных деталях, касающихся внутренних типов данных платформы .NET 5 и объявления переменных. Вы будете манипулировать текстовыми данными с применением типов `System.String` и `System.Text.StringBuilder`. Кроме того, вы исследуете итерационные конструкции и конструкции принятия решений, сопоставление с образцом, сужающие и расширяющие операции и ключевое слово `unchecked`.

Глава 4. Главные конструкции программирования на C#: часть 2

В этой главе завершается исследование ключевых аспектов C#, начиная с создания и манипулирования массивами данных. Затем вы узнаете, как конструировать перегруженные методы типов и определять параметры с применением ключевых слов `out`, `ref` и `params`. Также вы изучите типы перечислений, структуры и типы, допускающие `null`, плюс уясните отличие между типами значений и ссылочными типами. Наконец, вы освоите кортежи — средство, появившееся в C# 7 и обновленное в C# 8.

Часть III. Объектно-ориентированное программирование на C#

В этой части вы изучите основные конструкции языка C#, включая детали объектно-ориентированного программирования. Здесь вы научитесь обрабатывать исключения времени выполнения и взаимодействовать со строго типизированными интерфейсами. Вы также узнаете о времени существования объектов и сборке мусора.

Глава 5. Инкапсуляция

В этой главе начинается рассмотрение концепций объектно-ориентированного программирования (ООП) на языке C#. После представления главных принципов ООП (инкапсуляции, наследования и полиморфизма) будет показано, как строить надежные типы классов с использованием конструкторов, свойств, статических членов, констант и полей только для чтения. Вы также узнаете об определениях частичных типов, синтаксисе инициализации объектов и автоматических свойств, а в заключение главы будут рассматриваться типы записей, появившиеся в C# 9.0.

Глава 6. Наследование и полиморфизм

Здесь вы ознакомитесь с оставшимися главными принципами ООП (наследованием и полиморфизмом), которые позволяют создавать семейства связанных типов классов. Вы узнаете о роли виртуальных и абстрактных методов (и абстрактных базовых классов), а также о природе полиморфных интерфейсов. Затем вы исследуете сопоставление с образцом посредством ключевого слова `is` и в заключение выясните роль первичного базового класса платформы .NET Core — `System.Object`.

Глава 7. Структурированная обработка исключений

В этой главе обсуждаются способы обработки в кодовой базе аномалий, возникающих во время выполнения, за счет использования структурированной обработки исключений. Вы узнаете не только о ключевых словах C#, которые дают возможность решать такие задачи (`try`, `catch`, `throw`, `when` и `finally`), но и о разнице между исключениями уровня приложения и уровня системы. Вдобавок в главе будет показано, как настроить инструмент Visual Studio на прерывание для всех исключений, чтобы отлаживать исключения, оставшиеся без внимания.

Глава 8. Работа с интерфейсами

Материал этой главы опирается на ваше понимание объектно-ориентированной разработки и посвящен программированию на основе интерфейсов. Здесь вы узнаете, каким образом определять классы и структуры, поддерживающие несколько линий поведения, обнаруживать такие линии поведения во время выполнения и выборочно скрывать какие-то из них с применением явной реализации интерфейсов. В дополнение к созданию специальных интерфейсов вы научитесь реализовывать стандартные интерфейсы, доступные внутри платформы .NET Core, и использовать их для построения объектов, которые могут сортироваться, копироваться, перечисляться и сравниваться.

Глава 9. Время существования объектов

В финальной главе этой части исследуется управление памятью средой .NET Runtime с использованием сборщика мусора .NET Core. Вы узнаете о роли корневых элементов приложения, поколений объектов и типа `System.GC`. После пред-

ставления основ будут рассматриваться темы освобождаемых объектов (реализующих интерфейс `IDisposable`) и процесса финализации (с применением метода `System.Object.Finalize()`). В главе также описан класс `Lazy<T>`, позволяющий определять данные, которые не будут размещаться в памяти вплоть до поступления запроса со стороны вызывающего кода. Вы увидите, что такая возможность очень полезна, когда нежелательно загромождать кучу объектами, которые в действительности программе не нужны.

Часть IV. Дополнительные конструкции программирования на C#

В этой части книги вы углубите знания языка C# за счет исследования нескольких более сложных (и важных) концепций. Здесь вы завершите ознакомление с системой типов .NET Core, изучив коллекции и обобщения. Вы также освоите несколько более сложных средств C# (такие как методы расширения, перегрузка операций, анонимные типы и манипулирование указателями). Затем вы узнаете о делегатах и лямбда-выражениях, взглянете на язык LINQ, а в конце части ознакомитесь с процессами и многопоточным/асинхронным программированием.

Глава 10. Коллекции и обобщения

В этой главе исследуется тема *обобщений*. Вы увидите, что программирование с обобщениями предлагает способ создания типов и членов типов, которые содержат заполнители, указываемые вызывающим кодом. По существу обобщения значительно улучшают производительность приложений и безопасность в отношении типов. Здесь не только описаны разнообразные обобщенные типы из пространства имен `System.Collections.Generic`, но также показано, каким образом строить собственные обобщенные методы и типы (с ограничениями и без).

Глава 11. Расширенные средства языка C#

В этой главе вы сможете углубить понимание языка C# за счет исследования нескольких расширенных приемов программирования. Здесь вы узнаете, как перегружать операции и создавать специальные процедуры преобразования (явного и неявного) для типов. Вы также научитесь строить и взаимодействовать с индексаторами типов и работать с расширяющими методами, анонимными типами, частичными методами и указателями C#, используя контекст небезопасного кода.

Глава 12. Делегаты, события и лямбда-выражения

Целью этой главы является прояснение типа делегата. Выражаясь просто, делегат .NET Core представляет собой объект, который указывает на определенные методы в приложении. С помощью делегатов можно создавать системы, которые позволяют многочисленным объектам участвовать в двухстороннем взаимодействии. После исследования способов применения делегатов .NET Core вы ознакомитесь с ключевым словом `event` языка C#, которое упрощает манипулирование низкоуровневыми делегатами в коде. В завершение вы узнаете о роли лямбда-операции C# (=>), а также о связи между делегатами, анонимными методами и лямбда-выражениями.

Глава 13. LINQ to Objects

В этой главе начинается исследование языка интегрированных запросов (LINQ). Язык LINQ дает возможность строить строго типизированные выражения запро-

сов, которые могут применяться к многочисленным целевым объектам LINQ для манипулирования данными в самом широком смысле этого слова. Здесь вы изучите API-интерфейс LINQ to Objects, который позволяет применять выражения LINQ к контейнерам данных (например, массивам, коллекциям и специальным типам). Приведенная в главе информация будет полезна позже в книге при рассмотрении других API-интерфейсов.

Глава 14. Процессы, домены приложений и контексты загрузки

Опираясь на хорошее понимание вами сборок, в этой главе подробно раскрывается внутреннее устройство загруженной исполняемой сборки .NET Core. Целью главы является иллюстрация отношений между процессами, доменами приложений и контекстными границами. Упомянутые темы формируют основу для главы 15, где будет исследоваться конструирование многопоточных приложений.

Глава 15. Многопоточное, параллельное и асинхронное программирование

Эта глава посвящена построению многопоточных приложений. В ней демонстрируются приемы, которые можно использовать для написания кода, безопасного к потокам. Глава начинается с краткого напоминания о том, что собой представляет тип делегата .NET Core, и объяснения внутренней поддержки делегата для асинхронного вызова методов. Затем рассматриваются типы из пространства имен `System.Threading` и библиотека параллельных задач (Task Parallel Library — TPL). С применением TPL разработчики могут строить приложения .NET Core, которые распределяют рабочую нагрузку по всем доступным процессорам в исключительно простой манере. В главе также раскрыта роль API-интерфейса Parallel LINQ, который предлагает способ создания запросов LINQ, масштабируемых среди множества процессорных ядер. В завершение главы исследуется создание неблокирующих вызовов с использованием ключевых слов `async/await`, введенных в версии C# 5, локальных функций и обобщенных возвращаемых типов `async`, появившихся в версии C# 7, а также асинхронных потоков, добавленных в версии C# 8.

Часть V. Программирование с использованием сборок .NET Core

Эта часть книги посвящена деталям формата сборок .NET Core. Здесь вы узнаете не только о том, как развертывать и конфигурировать библиотеки кода .NET Core, но также о внутреннем устройстве двоичного образа .NET Core. Будет описана роль атрибутов .NET Core и распознавания информации о типе во время выполнения. Кроме того, объясняется роль исполняющей среды динамического языка (DLR) и ключевого слова `dynamic` языка C#. В последней главе части рассматривается синтаксис языка CIL и обсуждается роль динамических сборок.

Глава 16. Построение и конфигурирование библиотек классов

На самом высоком уровне термин "сборка" применяется для описания двоичного файла, созданного с помощью компилятора .NET Core. Однако в действительности понятие сборки намного шире. Вы научитесь создавать и развертывать сборки и узнаете, в чем отличие между библиотеками классов и консольными приложениями, а также между библиотеками классов .NET Core и .NET Standard. В конце главы раскрываются новые возможности, доступные в .NET 5, такие как однофайловое автономное развертывание.

Глава 17. Рефлексия типов, позднее связывание и программирование на основе атрибутов

В этой главе продолжается исследование сборок .NET Core. Здесь будет показано, как обнаруживать типы во время выполнения с использованием пространства имен `System.Reflection`. Посредством типов из упомянутого пространства имен можно строить приложения, способные считывать метаданные сборки на лету. Вы также узнаете, как загружать и создавать типы динамически во время выполнения с применением позднего связывания. Напоследок в главе обсуждается роль атрибутов .NET Core (стандартных и специальных). Для закрепления материала в главе демонстрируется построение расширяемого приложения с подключаемыми оснастками.

Глава 18. Динамические типы и среда DLR

В версии .NET 4.0 появился новый аспект исполняющей среды .NET, который называется *исполняющей средой динамического языка* (DLR). Используя DLR и ключевое слово `dynamic` языка C#, можно определять данные, которые в действительности не будут распознаваться вплоть до времени выполнения. Такие средства существенно упрощают решение ряда сложных задач программирования для .NET Core. В этой главе вы ознакомитесь со сценариями применения динамических данных, включая использование API-интерфейсов рефлексии .NET Core и взаимодействие с унаследованными библиотеками COM с минимальными усилиями.

Глава 19. Язык CIL и роль динамических сборок

В последней главе этой части преследуется двойная цель. В первой половине главы рассматривается синтаксис и семантика языка CIL, а во второй — роль пространства имен `System.Reflection.Emit`. Типы из указанного пространства имен можно применять для построения ПО, которое способно генерировать сборки .NET Core в памяти во время выполнения. Формально сборки, которые определяются и выполняются в памяти, называются *динамическими сборками*.

Часть VI. Работа с файлами, сериализация объектов и доступ к данным

К настоящему моменту вы уже должны хорошо ориентироваться в языке C# и в подробностях формата сборок .NET Core. В данной части книги ваши знания расширяются исследованием нескольких часто используемых служб, которые можно обнаружить внутри библиотек базовых классов, включая файловый ввод-вывод, сериализация объектов и доступ к базам данных посредством ADO.NET.

Глава 20. Файловый ввод-вывод и сериализация объектов

Пространство имен `System.IO` позволяет взаимодействовать со структурой файлов и каталогов машины. В этой главе вы узнаете, как программно создавать (и удалять) систему каталогов. Вы также научитесь перемещать данные между различными потоками (например, файловыми, строковыми и находящимися в памяти). Кроме того, в главе рассматриваются службы сериализации объектов в формат XML и JSON платформы .NET Core. Сериализация позволяет сохранять состояние объекта (или набора связанных объектов) в потоке для последующего использования. Десериализация представляет собой процесс извлечения объекта из потока в память с целью потребления внутри приложения.

Глава 21. Доступ к данным с помощью ADO.NET

Эта глава посвящена доступу к данным с использованием ADO.NET — API-интерфейса доступа к базам данных для приложений .NET Core. В частности, здесь рассматривается роль поставщиков данных .NET Core и взаимодействие с реляционной базой данных с применением инфраструктуры ADO.NET, которая представлена объектами подключений, объектами команд, объектами транзакций и объектами чтения данных. Кроме того, в главе начинается создание уровня доступа к данным `AutoLot`, который будет расширен в главах 22 и 23.

Часть VII. Entity Framework Core

У вас уже есть четкое представление о языке C# и деталях формата сборок .NET Core. В этой части вы узнаете о распространенных службах, реализованных внутри библиотек базовых классов, в числе которых файловый ввод-вывод, доступ к базам данных с использованием ADO.NET и доступ к базам данных с применением Entity Framework Core.

Глава 22. Введение в Entity Framework Core

В этой главе рассматривается инфраструктура Entity Framework (EF) Core, которая представляет собой систему объектно-реляционного отображения (ORM), построенную поверх ADO.NET. Инфраструктура EF Core предлагает способ написания кода доступа к данным с использованием строго типизированных классов, напрямую отображаемых на бизнес-модель. Здесь вы освоите строительные блоки EF Core, включая `DbContext`, сущности, специализированный класс коллекции `DbSet<T>` и `DbChangeTracker`. Затем вы узнаете о выполнении запросов, отслеживаемых и неотслеживаемых сущностях, а также о других примечательных возможностях EF Core. В заключение рассматривается глобальный инструмент EF Core для интерфейса командной строки .NET Core (CLI).

Глава 23. Построение уровня доступа к данным с помощью Entity Framework Core

В этой главе создается уровень доступа к данным `AutoLot`. Глава начинается с построения шаблонов сущностей и производного от `DbContext` класса для базы данных `AutoLot` из главы 21. Затем подход "сначала база данных" меняется на подход "сначала код". Сущности обновляются до своей финальной версии, после чего создается и выполняется миграция, чтобы обеспечить соответствие сущностям. Последнее изменение базы данных заключается в создании миграции для хранимой процедуры из главы 21 и нового представления базы данных. В целях инкапсуляции кода добавляются хранилища данных, и затем организуется процесс инициализации данных. Наконец, проводится испытание уровня доступа к данным с использованием инфраструктуры xUnit для автоматизированного интеграционного тестирования.

Часть VIII. Разработка клиентских приложений для Windows

Первоначальный API-интерфейс для построения графических пользовательских интерфейсов настольных приложений, поддерживаемый платформой .NET, назывался Windows Forms. Хотя он по-прежнему доступен, в версии .NET 3.0 программистам был предложен API-интерфейс под названием Windows Presentation Foundation (WPF). В отличие от Windows Forms эта инфраструктура для построения пользовательских

интерфейсов объединяет в единую унифицированную модель несколько основных служб, включая привязку данных, двумерную и трехмерную графику, анимацию и форматированные документы. Все это достигается с использованием декларативной грамматики разметки, которая называется расширяемым языком разметки приложений (XAML). Более того, архитектура элементов управления WPF предлагает легкий способ радикального изменения внешнего вида и поведения типового элемента управления с применением всего лишь правильно оформленной разметки XAML.

Глава 24. Введение в Windows Presentation Foundation и XAML

Эта глава начинается с исследования мотивации создания WPF (с учетом того, что в .NET уже существовала инфраструктура для разработки графических пользовательских интерфейсов настольных приложений). Затем вы узнаете о синтаксисе XAML и ознакомитесь с поддержкой построения приложений WPF в Visual Studio.

Глава 25. Элементы управления, компоновки, события и привязка данных в WPF

В этой главе будет показано, как работать с элементами управления и диспетчерами компоновки, предлагаемыми WPF. Вы узнаете, каким образом создавать системы меню, окна с разделителями, панели инструментов и строки состояния. Также в главе рассматриваются API-интерфейсы (и связанные с ними элементы управления), входящие в состав WPF, в том числе Ink API, команды, маршрутизируемые события, модель привязки данных и свойства зависимости.

Глава 26. Службы визуализации графики WPF

Инфраструктура WPF является API-интерфейсом, интенсивно использующим графику, и с учетом этого WPF предоставляет три подхода к визуализации графических данных: фигуры, рисунки и геометрические объекты, а также визуальные объекты. В настоящей главе вы ознакомитесь с каждым подходом и попутно изучите несколько важных графических примитивов (например, кисти, перья и трансформации). Кроме того, вы узнаете, как встраивать векторные изображения в графику WPF и выполнять операции проверки попадания в отношении графических данных.

Глава 27. Ресурсы, анимация, стили и шаблоны WPF

В этой главе освещены важные (и взаимосвязанные) темы, которые позволят углубить знания API-интерфейса WPF. Первым делом вы изучите роль логических ресурсов. Система логических ресурсов (также называемых объектными ресурсами) предлагает способ именования и ссылки на часто используемые объекты внутри приложения WPF. Затем вы узнаете, каким образом определять, выполнять и управлять анимационной последовательностью. Вы увидите, что применение анимации WPF не ограничивается видеоиграми или мультимедиа-приложениями. В завершение главы вы ознакомитесь с ролью стилей WPF. Подобно веб-странице, использующей CSS или механизм тем ASP.NET, приложение WPF может определять общий вид и поведение для целого набора элементов управления.

Глава 28. Уведомления WPF, проверка достоверности, команды и MVVM

Эта глава начинается с исследования трех основных возможностей инфраструктуры WPF: уведомлений, проверки достоверности и команд. В разделе, в котором рассматриваются уведомления, вы узнаете о наблюдаемых моделях и коллекциях, а

также о том, как они поддерживают данные приложения и пользовательский интерфейс в синхронизированном состоянии. Затем вы научитесь создавать специальные команды для инкапсуляции кода. В разделе, посвященном проверке достоверности, вы ознакомитесь с несколькими механизмами проверки достоверности, которые доступны для применения в приложениях WPF. Глава завершается исследованием паттерна "модель-представление-модель представления" (MVVM) и созданием приложения, демонстрирующего паттерн MVVM в действии.

Часть IX. ASP.NET Core

Эта часть посвящена построению веб-приложений с применением инфраструктуры ASP.NET Core, которую можно использовать для создания веб-приложений и служб REST.

Глава 29. Введение в ASP.NET Core

В этой главе обсуждается инфраструктура ASP.NET Core и паттерн MVC. Сначала объясняются функциональные средства, перенесенные в ASP.NET Core из классических инфраструктур ASP.NET MVC и Web API, в том числе контроллеры и действия, привязка моделей, маршрутизация и фильтры. Затем рассматриваются новые функциональные средства, появившиеся в ASP.NET Core, включая внедрение зависимостей, готовность к взаимодействию с облачными технологиями, осведомленная о среде система конфигурирования, шаблоны развертывания и конвейер обработки запросов HTTP. Наконец, в главе создаются два проекта ASP.NET Core, которые будут закончены в последующих двух главах, демонстрируются варианты запуска приложений ASP.NET Core и начинается процесс конфигурирования этих двух проектов ASP.NET Core.

Глава 30. Создание служб REST с помощью ASP.NET Core

В этой главе завершается создание приложения REST-службы ASP.NET Core. Первым делом демонстрируются разные механизмы возвращения клиенту результатов JSON и встроенная поддержка приложений служб, обеспечиваемая атрибутом `ApiController`. Затем добавляется пакет Swagger/OpenAPI, чтобы предоставить платформу для тестирования и документирования службы. В конце главы создаются контроллеры для приложения и фильтр исключений.

Глава 31. Создание приложений MVC с помощью ASP.NET Core

В последней главе книги заканчивается рассмотрение ASP.NET Core и работа над веб-приложением MVC. Сначала подробно обсуждаются представления и механизм представлений Razor, включая компоновки и частичные представления. Затем исследуются вспомогательные функции дескрипторов, а также управление библиотеками клиентской стороны и пакетирование/минификация этих библиотек. Далее завершается построение класса `CarsController` и его представлений вместе со вспомогательными функциями дескрипторов. В управляемое данными меню добавляется компонент представления и рассматривается шаблон параметров. Наконец, создается оболочка для службы клиента HTTP, а класс `CarsController` обновляется с целью использования службы ASP.NET Core вместо уровня доступа к данным `AutoLot`.

Ждем ваших отзывов!

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

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

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

Наши электронные адреса:

E-mail: `info.dialektika@gmail.com`

WWW: `http://www.dialektika.com`

ЧАСТЬ I

Язык программирования C# и платформа .NET 5

ГЛАВА 1

Введение в C# и .NET (Core) 5

Платформа Microsoft .NET и язык программирования C# впервые были представлены приблизительно в 2002 году и быстро стали главной опорой современной индустрии разработки программного обеспечения. Платформа .NET позволяет большому числу языков программирования (включая C#, VB.NET и F#) взаимодействовать друг с другом. На программу, написанную на C#, может ссылаться другая программа, написанная на VB.NET. Такая способность к взаимодействию более подробно обсуждается позже в главе.

В 2016 году компания Microsoft официально выпустила инфраструктуру .NET Core. Подобно .NET инфраструктура .NET Core позволяет языкам взаимодействовать друг с другом (хотя поддерживает ограниченное количество языков). Что более важно, новая инфраструктура способна функционировать не только под управлением операционной системы Windows, но также может запускаться (и позволять разрабатывать приложения) в средах iOS и Linux. Такая независимость от платформы открыла язык C# для гораздо большего числа разработчиков. Несмотря на то что межплатформенное использование C# поддерживалось и до выхода .NET Core, это делалось через ряд других инфраструктур, таких как проект Mono.

На заметку! Возможно, вас заинтересовало наличие круглых скобок в названии главы. С выходом .NET 5 часть "Core" в имени была отброшена с целью указания на то, что эта версия является унификацией всей платформы .NET. Но все же ради ясности повсюду в книге будут применяться термины .NET Core и .NET Framework.

---

10 ноября 2020 года компания Microsoft выпустила C# 9 и .NET 5. Как и C# 8, версия C# 9 привязана к определенной версии инфраструктуры и будет функционировать только под управлением .NET 5.0 и последующих версий. Привязка версии языка к версии инфраструктуры давала команде разработчиков C# свободу в плане ввода новых средств в C#, которые в противном случае не удалось бы добавить из-за ограничений инфраструктуры.

Во введении книги отмечалось, что при ее написании преследовались две цели. Первая из них — предоставление читателям глубокого и подробного описания синтаксиса и семантики языка C#. Вторая (не менее важная) цель — иллюстрация использования многочисленных API-интерфейсов .NET Core. В перечень рассматриваемых тем входят доступ к базам данных с помощью ADO.NET и Entity Framework (EF) Core, построение пользовательских интерфейсов посредством Windows Presentation Foundation (WPF), а также создание веб-служб REST и веб-приложений с применением ASP.NET Core. Как говорят, пеший поход длиной тысячу километров начинается с первого шага, который и будет сделан в настоящей главе.

Первая глава закладывает концептуальную основу для успешного освоения остального материала книги. Здесь вы найдете высокоуровневое обсуждение нескольких связанных с .NET тем, таких как сборки, общий промежуточный язык (Common Intermediate Language — CIL) и оперативная (Just-In-Time — JIT) компиляция. В дополнение к предварительному обзору ряда ключевых слов C# вы узнаете о взаимоотношениях между разнообразными компонентами .NET Core. Сюда входит исполняющая среда .NET Runtime, которая объединяет общеязыковую исполняющую среду .NET Core (.NET Core Common Language Runtime — CoreCLR) и библиотеки .NET Core (.NET Core Libraries — CoreFX) в единую кодовую базу, общая система типов (Common Type System — CTS), общеязыковая спецификация (Common Language Specification — CLS) и .NET Standard.

Кроме того, в главе представлен обзор функциональности, поставляемой в библиотеках базовых классов .NET Core, для обозначения которых иногда применяется аббревиатура BCL (base class library — библиотека базовых классов). Вы кратко ознакомитесь с независимой от языка и платформы природой .NET Core. Как несложно догадаться, многие затронутые здесь темы будут более детально исследоваться в оставшихся главах книги.

На заметку! Многие средства, рассматриваемые в настоящей главе (и повсюду в книге), также присутствуют в первоначальной инфраструктуре .NET Framework. В этой книге всегда будут использоваться термины "инфраструктура .NET Core" и "исполняющая среда .NET Core", а не общий термин ".NET", чтобы четко указывать, какие средства поддерживаются в .NET Core.

Некоторые основные преимущества инфраструктуры .NET Core

Инфраструктура .NET Core представляет собой программную платформу для построения веб-приложений и систем на основе служб, функционирующих под управлением операционных систем Windows, iOS и Linux, а также приложений Windows Forms и WPF для Windows. Ниже приведен краткий перечень основных средств, предлагаемых .NET Core.

- *Возможность взаимодействия с существующим кодом.* Несомненно, это очень полезно. Существующее программное обеспечение .NET Framework может взаимодействовать с более новым программным обеспечением .NET Core. Обратное взаимодействие тоже возможно через .NET Standard.
- *Поддержка многочисленных языков программирования.* Приложения .NET Core могут создаваться с использованием языков программирования C#, F# и VB.NET (при этом C# и F# являются основными языками для ASP.NET Core).
- *Общий исполняющий механизм, разделяемый всеми языками .NET Core.* Одним из аспектов такого механизма является наличие четко определенного набора типов, которые способен опознавать каждый язык .NET Core.
- *Языковая интеграция.* В .NET Core поддерживается межъязыковое наследование, межъязыковая обработка исключений и межъязыковая отладка кода. Например, можно определить базовый класс в C# и расширить этот тип в VB.NET.

- *Обширная библиотека базовых классов.* Данная библиотека предоставляет тысячи предварительно определенных типов, которые позволяют строить библиотеки кода, простые терминальные приложения, графические настольные приложения и веб-сайты производственного уровня.
- *Упрощенная модель развертывания.* Библиотеки .NET Core не регистрируются в системном реестре. Более того, платформа .NET Core позволяет нескольким версиям инфраструктуры и приложения гармонично сосуществовать на одном компьютере.
- *Всесторонняя поддержка командной строки.* Интерфейс командной строки .NET Core (command-line interface — CLI) является межплатформенной цепочкой инструментов для разработки и пакетирования приложений .NET Core. Помимо стандартных инструментов, поставляемых в составе .NET Core SDK, могут быть установлены дополнительные инструменты.

Все перечисленные темы (и многие другие) будут подробно рассматриваться в последующих главах. Но сначала необходимо объяснить новый жизненный цикл поддержки для .NET Core.

Понятие жизненного цикла поддержки .NET Core

Версии .NET Core выходят гораздо чаще, нежели версии .NET Framework. Из-за обилия доступных выпусков может быть трудно не отставать, особенно в корпоративной среде разработки. Чтобы лучше определить жизненный цикл поддержки для выпусков, компания Microsoft приняла вариацию модели долгосрочной поддержки (Long-Term Support — LTS)[1], обычно применяемой современными инфраструктурами с открытым кодом.

Выпуски с поддержкой LTS — это крупные выпуски, которые будут поддерживаться в течение длительного периода времени. На протяжении своего срока службы они будут получать только критически важные и/или неразрушающие исправления. Перед окончанием срока службы версии LTS изменяются с целью сопровождения. Выпуски LTS инфраструктуры .NET Core будут поддерживаться для следующих периодов времени в зависимости от того, какой из них длиннее:

- три года после первоначального выпуска;
- один год технической поддержки после следующего выпуска LTS.

В Microsoft решили именовать выпуски LTS как Current (текущие), которые являются промежуточными выпусками между крупными выпусками LTS. Они поддерживаются на протяжении трех месяцев после следующего выпуска Current или LTS.

Как упоминалось ранее, версия .NET 5 вышла 10 ноября 2020 года. Она была выпущена как версия Current, а не LTS. Это значит, что поддержка .NET 5 прекратится через три месяца после выхода следующего выпуска. Версия .NET Core 3.1, выпущенная в декабре 2019 года, представляет собой версию LTS и полноценно поддерживается вплоть до 3 декабря 2022 года.

[1] https://ru.wikipedia.org/wiki/Долгосрочная_поддержка_программного_обеспечения

На заметку! Следующим запланированным выпуском .NET будет версия .NET 6, которая по графику должна появиться в ноябре 2021 года. В итоге получается примерно 15 месяцев поддержки .NET 5. Однако если в Microsoft решат выпустить исправления (скажем, .NET 5.1), тогда трехмесячный срок начнется с этого выпуска. Мы рекомендуем обдумать такую политику поддержки, когда вы будете выбирать версию для разработки производственных приложений. Важно понимать: речь не идет о том, что вы не должны использовать .NET 5. Мы всего лишь настоятельно советуем надлежащим образом разобраться в политике поддержки при выборе версий .NET (Core) для разработки производственных приложений.

Обязательно проверяйте политику поддержки для каждой новой выпущенной версии .NET Core. Наличие более высокого номера версии не обязательно означает, что она будет поддерживаться в течение длительного периода времени. Полное описание политики поддержки доступно по ссылке `https://dotnet.microsoft.com/platform/support-policy/dotnet-core`.

Предварительный обзор строительных блоков .NET Core (.NET Runtime, CTS и CLS)

Теперь, когда вы узнали кое-что об основных преимуществах, присущих .NET Core, давайте ознакомимся с ключевыми (и взаимосвязанными) компонентами, которые делают возможным все упомянутое ранее — Core Runtime (формально CoreCLR и CoreFX), CTS и CLS. С точки зрения программиста приложений платформу .NET Core можно воспринимать как исполняющую среду и обширную библиотеку базовых классов. Уровень исполняющей среды содержит набор минимальных реализаций, которые привязаны к конкретным платформам (Windows, iOS, Linux) и архитектурам (x86, x64, ARM), а также все базовые типы для .NET Core.

Еще одним строительным блоком .NET Core является *общая система типов* (CTS). Спецификация CTS полностью описывает все возможные типы данных и все программные конструкции, поддерживаемые исполняющей средой, указывает, каким образом эти сущности могут взаимодействовать друг с другом, и как они представлены в формате метаданных .NET Core (дополнительную информацию о метаданных ищите далее в главе, а исчерпывающие сведения — в главе 17).

Важно понимать, что отдельно взятый язык .NET Core может не поддерживать абсолютно все функциональные средства, определяемые спецификацией CTS. Существует родственная *общеязыковая спецификация* (CLS), где описано подмножество общих типов и программных конструкций, которое должны поддерживать все языки программирования .NET Core. Таким образом, если вы строите типы .NET Core, открывающие доступ только к совместимым с CLS средствам, то можете быть уверены в том, что их смогут потреблять все языки .NET Core. И наоборот, если вы применяете тип данных или программную конструкцию, которая выходит за границы CLS, тогда не сможете гарантировать, что каждый язык программирования .NET Core окажется способным взаимодействовать с вашей библиотекой кода .NET Core. К счастью, как вы увидите далее в главе, компилятору C# довольно просто сообщить о необходимости проверки всего кода на предмет совместимости с CLS.

Роль библиотек базовых классов

Инфраструктура .NET Core также предоставляет набор библиотек базовых классов (BCL), которые доступны всем языкам программирования .NET Core. Библиотеки базовых классов не только инкапсулируют разнообразные примитивы вроде потоков, файлового ввода-вывода, систем визуализации графики и механизмов взаимодействия с разнообразными внешними устройствами, но вдобавок обеспечивают поддержку для многочисленных служб, требуемых большинством реальных приложений.

В библиотеках базовых классов определены типы, которые можно применять для построения программного приложения любого вида и для компонентов приложений, взаимодействующих друг с другом.

Роль .NET Standard

Даже с учетом выхода .NET 5.0 количество библиотек базовых классов в .NET Framework намного превышает количество библиотек подобного рода в .NET Core. Учитывая 14-летнее преимущество .NET Framework над .NET Core, ситуация вполне объяснима. Такое несоответствие создает проблемы при попытке использования кода .NET Framework с кодом .NET Core. Решением (и требованием) для взаимодействия .NET Framework/.NET Core является стандарт .NET Standard.

.NET Standard — это спецификация, определяющая доступность API-интерфейсов .NET и библиотек базовых классов, которые должны присутствовать в каждой реализации. Стандарт обладает следующими характеристиками:

- определяет унифицированный набор API-интерфейсов BCL для всех реализаций .NET, которые должны быть созданы независимо от рабочей нагрузки;
- позволяет разработчикам производить переносимые библиотеки, пригодные для потребления во всех реализациях .NET, с использованием одного и того же набора API-интерфейсов;
- сокращает или даже устраняет условную компиляцию общего исходного кода API-интерфейсов .NET, оставляя ее только для API-интерфейсов операционной системы.

В таблице, приведенной в документации от Microsoft (`https://docs.microsoft.com/ru-ru/dotnet/standard/net-standard`), указаны минимальные версии реализаций, которые поддерживают каждый стандарт .NET Standard. Она полезна в случае применения предшествующих версий C#. Тем не менее, версия C# 9 будет функционировать только в среде .NET 5.0 (или выше) либо .NET Standard 2.1, а стандарт .NET Standard 2.1 не является доступным для .NET Framework.

Что привносит язык C#

Синтаксис языка программирования C# выглядит *очень* похожим на синтаксис языка Java. Однако называть C# клоном Java неправильно. В действительности и C#, и Java являются членами семейства языков программирования, основанных на C (например, C, Objective-C, C++), поэтому они разделяют сходный синтаксис.

Правда заключается в том, что многие синтаксические конструкции C# смоделированы в соответствии с разнообразными аспектами языков VB и C++. Например, подобно VB язык C# поддерживает понятия свойств класса (как противоположность традиционным методам извлечения и установки) и необязательных параметров.

Подобно C++ язык C# позволяет перегружать операции, а также создавать структуры, перечисления и функции обратного вызова (посредством делегатов).

Более того, по мере проработки материала книги вы очень скоро заметите, что C# поддерживает средства, такие как лямбда-выражения и анонимные типы, которые традиционно встречаются в различных языках функционального программирования (например, LISP или Haskell). Вдобавок с появлением технологии LINQ (*Language Integrated Query* — язык интегрированных запросов) язык C# стал поддерживать конструкции, которые делают его довольно-таки уникальным в мире программирования. Но, несмотря на все это, наибольшее влияние на него оказали именно языки, основанные на C.

Поскольку C# — гибрид из нескольких языков, он является таким же синтаксически чистым, как Java (если не чище), почти настолько же простым, как VB, и практически таким же мощным и гибким, как C++. Ниже приведен неполный перечень ключевых особенностей языка C#, которые характерны для всех его версий.

- Указатели необязательны! В программах на C# обычно не возникает потребности в прямых манипуляциях указателями (хотя в случае абсолютной необходимости можно опуститься и на уровень указателей, как объясняется в главе 11).
- Автоматическое управление памятью посредством сборки мусора. С учетом этого в C# не поддерживается ключевое слово вроде `delete`.
- Формальные синтаксические конструкции для классов, интерфейсов, структур, перечислений и делегатов.
- Аналогичная языку C++ возможность перегрузки операций для специальных типов без особой сложности.
- Поддержка программирования на основе атрибутов. Разработка такого вида позволяет аннотировать типы и их члены для дополнительного уточнения их поведения. Например, если пометить метод атрибутом `[Obsolete]`, то при попытке его использования программисты увидят ваше специальное предупреждение.

C# 9 уже является мощным языком, который в сочетании с .NET Core позволяет строить широкий спектр приложений разнообразных видов.

Основные средства в предшествующих выпусках

С выходом версии .NET 2.0 (примерно в 2005 году) язык программирования C# был обновлен с целью поддержки многочисленных новых функциональных возможностей, наиболее значимые из которых перечислены далее.

- Возможность создания обобщенных типов и обобщенных членов. Применяя обобщения, можно писать очень эффективный и безопасный к типам код, который определяет множество заполнителей, указываемых во время взаимодействия с обобщенными элементами.
- Поддержка анонимных методов, которые позволяют предоставлять встраиваемую функцию везде, где требуется тип делегата.
- Возможность определения одиночного типа в нескольких файлах кода (или при необходимости в виде представления в памяти) с использованием ключевого слова `partial`.

В версии .NET 3.5 (вышедшей приблизительно в 2008 году) к языку программирования C# была добавлена дополнительная функциональность, в том числе следующие средства.

- Поддержка строго типизированных запросов (например, LINQ), применяемых для взаимодействия с разнообразными формами данных. Вы впервые встретите запросы LINQ в главе 13.
- Поддержка анонимных типов, позволяющая моделировать на лету в коде устройство типа, а не его поведение.
- Возможность расширения функциональности существующего типа (не создавая его подклассы) с использованием расширяющих методов.
- Включение лямбда-операции (=>), которая еще больше упрощает работу с типами делегатов .NET.
- Новый синтаксис инициализации объектов, позволяющий устанавливать значения свойств во время создания объекта.

В версии .NET 4.0 (выпущенной в 2010 году) язык C# снова был дополнен рядом средств, которые указаны ниже.

- Поддержка необязательных параметров и именованных аргументов в методах.
- Поддержка динамического поиска членов во время выполнения через ключевое слово `dynamic`. Как будет показано в главе 19, это обеспечивает универсальный подход к вызову членов на лету независимо от инфраструктуры, в которой они реализованы (COM, IronRuby, IronPython или службы рефлексии .NET).
- Работа с обобщенными типами стала намного понятнее, учитывая возможность легкого отображения обобщенных данных на универсальные коллекции `System.Object` через ковариантность и контравариантность.

В выпуске .NET 4.5 язык C# обрел пару новых ключевых слов (`async` и `await`), которые значительно упрощают многопоточное и асинхронное программирование. Если вы работали с предшествующими версиями C#, то можете вспомнить, что вызов методов через вторичные потоки требовал довольно большого объема малопонятного кода и применения разнообразных пространств имен .NET. Учитывая то, что теперь в C# поддерживаются языковые ключевые слова, которые автоматически устраняют эту сложность, процесс вызова методов асинхронным образом оказывается почти настолько же легким, как их вызов в синхронной манере. Данные темы детально раскрываются в главе 15.

Версия C# 6 появилась в составе .NET 4.6 и получила несколько мелких средств, которые помогают упростить кодовую базу. Ниже представлен краткий обзор ряда средств, введенных в C# 6.

- Встраиваемая инициализация для автоматических свойств, а также поддержка автоматических свойств, предназначенных только для чтения.
- Реализация однострочных методов с использованием лямбда-операции C#.
- Поддержка статического импортирования для предоставления прямого доступа к статическим членам внутри пространства имен.
- `null`-условная операция, которая помогает проверять параметры на предмет `null` в реализации метода.

- Новый синтаксис форматирования строк, называемый интерполяцией строк.
- Возможность фильтрации исключений с применением нового ключевого слова `when`.
- Использование `await` в блоках `catch` и `finally`.
- Выражения `nameof` для возвращения строкового представления символов.
- Инициализаторы индексов.
- Улучшенное распознавание перегруженных версий.

В версии C# 7, выпущенной вместе с .NET 4.7 в марте 2017 года, были введены дополнительные средства для упрощения кодовой базы и добавлено несколько более значительных средств (вроде кортежей и ссылочных локальных переменных, а также возвращаемых ссылочных значений), которые разработчики просили включить довольно долгое время. Вот краткий обзор новых средств C# 7.

- Объявление переменных `out` как встраиваемых аргументов.
- Локальные функции.
- Дополнительные члены, сжатые до выражений.
- Обобщенные асинхронные возвращаемые типы.
- Новые маркеры для улучшения читабельности числовых констант.
- Легковесные неименованные типы (называемые кортежами), которые содержат множество полей.
- Обновления логического потока с применением сопоставления с типом вдобавок к проверке значений (сопоставлению с образцом).
- Возвращение ссылки на значение вместо только самого значения (ссылочные локальные переменные и возвращаемые ссылочные значения).
- Введение легковесных одноразовых переменных (называется отбрасыванием).
- Выражения `throw`, позволяющие размещать конструкцию `throw` в большем числе мест — в условных выражениях, лямбда-выражениях и др.

С версией C# 7 связаны два младших выпуска, которые добавили следующие средства.

- Возможность иметь асинхронный метод `Main()` программы.
- Новый литерал `default`, который делает возможной инициализацию любого типа.
- Устранение проблемы при сопоставлении с образцом, которая препятствовала использованию обобщений в этом новом средстве сопоставления с образцом.
- Подобно анонимным методам имена кортежей могут выводиться из проекции, которая их создает.
- Приемы для написания безопасного и эффективного кода, сочетание синтаксических улучшений, которые позволяют работать с типами значений, применяя ссылочную семантику.
- За именованными аргументами могут следовать позиционные аргументы.

- Числовые литералы теперь могут иметь ведущие символы подчеркивания перед любыми печатаемыми цифрами.
- Модификатор доступа `private protected` делает возможным доступ для производных классов в той же самой сборке.
- Результатом условного выражения (`?:`) теперь может быть ссылка.

Кроме того, в этом издании книги к заголовкам разделов добавляются указания "(нововведение в версии 7.x)" и "(обновление в версии 7.x)", чтобы облегчить поиск изменений в языке по сравнению с предыдущей версией. Буква "x" означает младшую версию C# 7, такую как 7.1.

В версии C# 8, ставшей доступной 23 сентября 2019 года в рамках .NET Core 3.0, были введены дополнительные средства для упрощения кодовой базы и добавлен ряд более значимых средств (вроде кортежей, а также ссылочных локальных переменных и возвращаемых значений), которые разработчики просили включить в спецификацию языка в течение довольно долгого времени.

Версия C# 8 имеет два младших выпуска, которые добавили следующие средства:

- члены, допускающие только чтение, для структур;
- стандартные члены интерфейса;
- улучшения сопоставления с образцом;
- использование объявлений;
- статические локальные функции;
- освобождаемые ссылочные структуры;
- ссылочные типы, допускающие значение `null`;
- асинхронные потоки;
- индексы и диапазоны;
- присваивание с объединением с `null`;
- неуправляемые сконструированные типы;
- применение `stackalloc` во вложенных выражениях;
- усовершенствование интерполированных дословных строк.

Новые средства в C# 8 обозначаются как "(нововведение в версии 8)" в заголовках разделов, которые им посвящены, а обновленные средства помечаются как "(обновление в версии 8.0)".

Новые средства в C# 9

В версию C# 9, выпущенную 10 ноября 2020 года в составе .NET 5, добавлены следующие средства:

- записи;
- средства доступа только для инициализации;
- операторы верхнего уровня;
- улучшения сопоставления с образцом;
- улучшения производительности для взаимодействия;
- средства "подгонки и доводки";
- поддержка для генераторов кода.

Новые средства в C# 9 обозначаются как "(нововведение в версии 9.0)" в заголовках разделов, которые им посвящены, а обновленные средства помечаются как "(обновление в версии 9.0)".

Сравнение управляемого и неуправляемого кода

Важно отметить, что язык C# может применяться только для построения программного обеспечения, которое функционирует под управлением исполняющей среды .NET Core (вы никогда не будете использовать C# для создания COM-сервера или неуправляемого приложения в стиле C/C++). Выражаясь официально, для обозначения кода, ориентированного на исполняющую среду .NET Core, используется термин *управляемый код*. Двоичный модуль, который содержит управляемый код, называется *сборкой* (сборки более подробно рассматриваются далее в главе). И наоборот, код, который не может напрямую обслуживаться исполняющей средой .NET Core, называется *неуправляемым кодом*.

Как упоминалось ранее, инфраструктура .NET Core способна функционировать в средах разнообразных операционных систем. Таким образом, вполне вероятно создавать приложение C# на машине Windows с применением Visual Studio и запускать его под управлением iOS с использованием исполняющей среды .NET Core. Кроме того, приложение C# можно построить на машине Linux с помощью Visual Studio Code и запускать его на машине Windows. С помощью Visual Studio для Mac на компьютере Mac можно разрабатывать приложения .NET Core, предназначенные для выполнения под управлением Windows, macOS или Linux.

Программа C# по-прежнему может иметь доступ к неуправляемому коду, но тогда она привяжет вас к специфической цели разработки и развертывания.

Использование дополнительных языков программирования, ориентированных на .NET Core

Имейте в виду, что C# — не единственный язык, который может применяться для построения приложений .NET Core. В целом приложения .NET Core могут строиться с помощью C#, Visual Basic и F#, которые представляют собой три языка, напрямую поддерживаемые Microsoft.

Обзор сборок .NET

Независимо от того, какой язык .NET Core выбран для программирования, важно понимать, что хотя двоичные модули .NET Core имеют такое же файловое расширение, как и неуправляемые двоичные компоненты Windows (`*.dll`), внутренне они устроены совершенно по-другому. В частности, двоичные модули .NET Core содержат не специфические, а независимые от платформы инструкции на *промежуточном языке* (Intermediate Language — IL) и метаданные типов.

На заметку! Язык IL также известен как промежуточный язык Microsoft (Microsoft Intermediate Language — MSIL) или общий промежуточный язык (Common Intermediate Language — CIL). Таким образом, при чтении литературы по .NET/.NET Core не забывайте о том, что IL, MSIL и CIL описывают в точности одну и ту же концепцию. В настоящей книге при ссылке на этот низкоуровневый набор инструкций будет применяться аббревиатура CIL.

Когда файл `*.dll` был создан с использованием компилятора .NET Core, результирующий большой двоичный объект называется *сборкой*. Все многочисленные детали, касающиеся сборок .NET Core, подробно рассматриваются в главе 16. Тем не менее, для упрощения текущего обсуждения вы должны усвоить четыре основных свойства нового файлового формата.

Во-первых, в отличие от сборок .NET Framework, которые могут быть файлами `*.dll` или `*.exe`, проекты .NET Core *всегда* компилируются в файл с расширением `.dll`, даже если проект является исполняемым модулем. Исполняемые сборки .NET Core выполняются с помощью команды `dotnet <имя_сборки>.dll`. Нововведение .NET Core 3.0 (и последующих версий) заключается в том, что команда `dotnet.exe` копирует файл в каталог сборки и переименовывает его на `<имя_сборки>.exe`. Запуск этой команды автоматически выполняет эквивалент `dotnet <имя_сборки>.exe`. Файл `*.exe` с именем вашего проекта фактически не относится к коду проекта; он является удобным сокращением для запуска вашего приложения.

Нововведением .NET 5 стало то, что ваше приложение может быть сведено до единственного файла, который запускается напрямую. Хотя такой единственный файл выглядит и действует подобно собственному исполняемому модулю в стиле C++, его преимущество заключается в пакетировании. Он содержит все файлы, необходимые для выполнения вашего приложения и потенциально даже саму исполняющую среду .NET 5! Но помните о том, что ваш код по-прежнему выполняется в управляемом контейнере, как если бы он был опубликован в виде множества файлов.

Во-вторых, сборка содержит код CIL, который концептуально похож на байт-код Java тем, что не компилируется в специфичные для платформы инструкции до тех пор, пока это не станет абсолютно необходимым. Обычно "абсолютная необходимость" наступает тогда, когда на блок инструкций CIL (такой как реализация метода) производится ссылка с целью его применения исполняющей средой .NET Core.

В-третьих, сборки также содержат *метаданные*, которые детально описывают характеристики каждого "типа" внутри двоичного модуля. Например, если имеется класс по имени `SportsCar` , то метаданные типа представляют такие детали, как базовый класс `SportsCar`, указывают реализуемые `SportsCar` интерфейсы (если есть) и дают полные описания всех членов, поддерживаемых типом `SportsCar`. Метаданные .NET Core всегда присутствуют внутри сборки и автоматически генерируются компилятором языка.

Наконец, в-четвертых, помимо инструкций CIL и метаданных типов сами сборки также описываются с помощью метаданных, которые официально называются *манифестом*. Манифест содержит информацию о текущей версии сборки, сведения о культуре (используемые для локализации строковых и графических ресурсов) и список ссылок на все внешние сборки, которые требуются для надлежащего функционирования. Разнообразные инструменты, которые можно применять для исследования типов, метаданных и манифестов сборок, рассматриваются в нескольких последующих главах.

Роль языка CIL

Теперь давайте займемся детальными исследованиями кода CIL, метаданных типов и манифеста сборки. Язык CIL находится выше любого набора инструкций, специфичных для конкретной платформы. Например, приведенный далее код C# моделирует простой калькулятор. Не углубляясь пока в подробности синтаксиса, обратите внимание на формат метода `Add()` в классе `Calc`.

```
// Calc.cs
using System;
namespace CalculatorExamples
{
  // Этот класс содержит точку входа приложения.
  class Program
  {
    static void Main(string[] args)
    {
      Calc c = new Calc();
      int ans = c.Add(10, 84);
      Console.WriteLine("10 + 84 is {0}.", ans);
      // Ожидать нажатия пользователем клавиши <Enter>
      // перед завершением работы.
      Console.ReadLine();
    }
  }
  // Калькулятор C#.
  class Calc
  {
    public int Add(int addend1, int addend2)
    {
      return addend1 + addend2;
    }
  }
}
```

Результатом компиляции такого кода будет файл `*.dll` сборки, который содержит манифест, инструкции CIL и метаданные, описывающие каждый аспект классов `Calc` и `Program`.

На заметку! В главе 2 будет показано, как использовать для компиляции файлов кода графические среды интегрированной разработки (integrated development environment — IDE), такие как Visual Studio Community.

Например, если вы выведете код IL из полученной сборки с помощью `ildasm.exe` (рассматривается чуть позже в главе), то обнаружите, что метод `Add()` был представлен в CIL следующим образом:

```
.method public hidebysig instance int32
        Add(int32 addend1, int32 addend2) cil managed
{
  // Code size       9 (0x9)
  // Размер кода     9 (0x9)
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  add
  IL_0004:  stloc.0
  IL_0005:  br.s     IL_0007
  IL_0007:  ldloc.0
  IL_0008:  ret
} // end of method Calc::Add      конец метода Calc::Add
```

Не беспокойтесь, если результирующий код CIL этого метода выглядит непонятным — в главе 19 будут описаны базовые аспекты языка программирования CIL. Важно понимать, что компилятор C# выпускает код CIL, а не инструкции, специфичные для платформы.

Теперь вспомните, что сказанное справедливо для всех компиляторов .NET. В целях иллюстрации создадим то же самое приложение на языке Visual Basic вместо C#:

```
' Calc.vb
Namespace CalculatorExample
  Module Program
    ' Этот класс содержит точку входа приложения.
    Sub Main(args As String())
      Dim c As New Calc
      Dim ans As Integer = c.Add(10, 84)
      Console.WriteLine("10 + 84 is {0}", ans)
      ' Ожидать нажатия пользователем клавиши <Enter>
      ' перед завершением работы.
      Console.ReadLine()
    End Sub
  End Module
  ' Калькулятор VB.NET.
  Class Calc
    Public Function Add(ByVal addend1 As Integer,
                        ByVal addend2 As Integer) As Integer
      Return addend1 + addend2
    End Function
  End Class
End Namespace
```

Просмотрев код CIL такого метода `Add()`, можно найти похожие инструкции (слегка скорректированные компилятором Visual Basic):

```
.method public instance int32 Add(int32 addend1,
                                   int32 addend2) cil managed
{
  // Code size      9 (0x9)
  // Размер кода    9 (0x9)
  .maxstack 2
  .locals init (int32 V_0)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  add.ovf
  IL_0004:  stloc.0
  IL_0005:  br.s     IL_0007

  IL_0007:  ldloc.0
  IL_0008:  ret
} // end of method Calc::Add
  // конец метода Calc::Add
```

В качестве финального примера ниже представлена та же самая простая программа `Calc`, разработанная на F# (еще одном языке .NET Core):

```
// Узнайте больше о языке F# на веб-сайте http://fsharp.org

// Calc.fs
open System

module Calc =
  let add addend1 addend2 =
    addend1 + addend2

[<EntryPoint>]
let main argv =
  let ans = Calc.add 10 84
  printfn "10 + 84 is %d" ans
  Console.ReadLine()
  0
```

Если вы просмотрите код CIL для метода `Add()`, то снова найдете похожие инструкции (слегка скорректированные компилятором F#).

```
.method public static int32 Add(int32 addend1,
                                 int32 addend2) cil managed
{
  .custom instance void [FSharp.Core]Microsoft.FSharp.Core.
CompilationArgumentCountsAttribute::.ctor(int32[]) = ( 01 00 02 00 00
00 01 00 00 00 01 00 00 00 00 00 )
  // Code size      4 (0x4)
  // Размер кода    4 (0x4)
  .maxstack 8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  add
  IL_0003:  ret
} // end of method Calc::'add'
  // конец метода Calc::'add'
```

Преимущества языка CIL

В этот момент вас может интересовать, какую выгоду приносит компиляция исходного кода в CIL, а не напрямую в специфичный набор инструкций. Одним из преимуществ является языковая интеграция. Как вы уже видели, все компиляторы .NET Core выпускают практически идентичные инструкции CIL. Следовательно, все языки способны взаимодействовать в рамках четко определенной "двоичной арены".

Более того, учитывая независимость от платформы языка CIL, сама инфраструктура .NET Core не зависит от платформы и обеспечивает те же самые преимущества, к которым так привыкли разработчики на Java (например, единую кодовую базу, функционирующую в средах многочисленных операционных систем). В действительности для языка C# предусмотрен международный стандарт. До выхода .NET Core существовало множество реализаций .NET для платформ, отличающихся от Windows, таких как Mono. Они по-прежнему доступны, хотя благодаря межплатформенной природе .NET Core потребность в них значительно снизилась.

Компиляция кода CIL в инструкции, специфичные для платформы

Поскольку сборки содержат инструкции CIL, а не инструкции, специфичные для платформы, перед применением код CIL должен компилироваться на лету.

Компонентом, который транслирует код CIL в содержательные инструкции центрального процессора (ЦП), является оперативный (JIT) компилятор (иногда называемый *jitter*). Для каждого целевого ЦП исполняющая среда .NET Core задействует JIT-компилятор, который оптимизирован под лежащую в основе платформу.

Скажем, если строится приложение .NET Core, предназначенное для развертывания на карманном устройстве (наподобие смартфона с iOS или Android), то соответствующий JIT-компилятор будет оснащен возможностями запуска в среде с ограниченным объемом памяти. С другой стороны, если сборка развертывается на внутреннем сервере компании (где память редко оказывается проблемой), тогда JIT-компилятор будет оптимизирован для функционирования в среде с большим объемом памяти. Таким образом, разработчики могут писать единственный блок кода, который способен эффективно транслироваться JIT-компилятором и выполняться на машинах с разной архитектурой.

Вдобавок при трансляции инструкций CIL в соответствующий машинный код JIT-компилятор будет кешировать результаты в памяти в манере, подходящей для целевой ОС. В таком случае, если производится вызов метода по имени `PrintDocument()`, то инструкции CIL компилируются в специфичные для платформы инструкции при первом вызове и остаются в памяти для более позднего использования. Благодаря этому при вызове метода `PrintDocument()` в следующий раз повторная компиляция инструкций CIL не понадобится.

Предварительная компиляция кода CIL в инструкции, специфичные для платформы

В .NET Core имеется утилита под названием `crossgen.exe`, которую вы можете использовать для предварительной компиляции JIT своего кода. К счастью, в .NET Core 3.0 возможность производить "готовые к запуску" сборки встроена в инфраструктуру. Более подробно об этом речь пойдет позже в книге.

Роль метаданных типов .NET Core

В дополнение к инструкциям CIL сборка .NET Core содержит полные и точные метаданные, которые описывают каждый определенный в двоичном модуле тип (например, класс, структуру, перечисление), а также члены каждого типа (скажем, свойства, методы, события). К счастью, за выпуск актуальных метаданных типов всегда отвечает компилятор, а не программист. Из-за того, что метаданные .NET Core настолько основательны, сборки являются целиком самоописательными сущностями.

Чтобы проиллюстрировать формат метаданных типов .NET Core, давайте взглянем на метаданные, которые были сгенерированы для исследуемого ранее метода `Add()` класса `Calc`, написанного на C# (метаданные для версии Visual Basic метода `Add()` похожи, так что будет исследоваться только версия C#):

```
TypeDef #2 (02000003)
-------------------------------------------------------
  TypDefName: CalculatorExamples.Calc (02000003)
  Flags     : [NotPublic] [AutoLayout] [Class] [AnsiClass]
[BeforeFieldInit] (00100000)
  Extends   : 0100000C [TypeRef] System.Object
  Method #1 (06000003)
  -----------------------------------------------------
```

```
MethodName : Add (06000003)
Flags      : [Public] [HideBySig] [ReuseSlot] (00000086)
RVA        : 0x00002090
ImplFlags  : [IL] [Managed] (00000000)
CallCnvntn : [DEFAULT]
hasThis
ReturnType: I4
2 Arguments
  Argument #1: I4
  Argument #2: I4
2 Parameters
  (1) ParamToken : (08000002) Name : addend1 flags: [none] (00000000)
  (2) ParamToken : (08000003) Name : addend2 flags: [none] (00000000)
```

Метаданные применяются многочисленными аспектами исполняющей среды .NET Core, а также разнообразными инструментами разработки. Например, средство IntelliSense, предоставляемое такими инструментами, как Visual Studio, стало возможным благодаря чтению метаданных сборки во время проектирования. Метаданные также используются различными утилитами просмотра объектов, инструментами отладки и самим компилятором C#. Бесспорно, метаданные являются принципиальной основой многочисленных технологий .NET Core, включая рефлексию, позднее связывание и сериализацию объектов. Роль метаданных .NET будет раскрыта в главе 17.

Роль манифеста сборки

Последний, но не менее важный момент: вспомните, что сборка .NET Core содержит также и метаданные, которые описывают ее саму (формально называемые *манифестом*). Помимо прочего манифест документирует все внешние сборки, которые требуются текущей сборке для ее корректного функционирования, номер версии сборки, информацию об авторских правах и т.д. Подобно метаданным типов за генерацию манифеста сборки всегда отвечает компилятор. Ниже представлены некоторые существенные детали манифеста, сгенерированного при компиляции показанного ранее в главе файла кода `Calc.cs` (ради краткости некоторые строки не показаны):

```
.assembly extern /*23000001*/ System.Runtime
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A ) // .?_....:
  .ver 5:0:0:0
}
.assembly extern /*23000002*/ System.Console
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A ) // .?_....:
  .ver 5:0:0:0
}
.assembly /*20000001*/ Calc.Cs
{
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module Calc.Cs.dll
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003          // WINDOWS_CUI
.corflags 0x00000001       // ILONLY
```

Выражаясь кратко, показанный манифест документирует набор внешних сборок, требуемых для `Calc.dll` (в директиве `.assembly extern`), а также разнообразные характеристики самой сборки (вроде номера версии и имени модуля). Полезность данных манифеста будет более подробно исследоваться в главе 16.

Понятие общей системы типов

Сборка может содержать любое количество различающихся типов. В мире .NET Core *тип* — это просто общий термин, применяемый для ссылки на член из набора {класс, интерфейс, структура, перечисление, делегат}. При построении решений на любом языке .NET Core почти наверняка придется взаимодействовать со многими такими типами. Например, в сборке может быть определен класс, реализующий некоторое количество интерфейсов. Возможно, метод одного из интерфейсов принимает перечисление в качестве входного параметра и возвращает вызывающему компоненту структуру.

Вспомните, что CTS является формальной спецификацией, которая документирует, каким образом типы должны быть определены, чтобы они могли обслуживаться .NET Runtime. Внутренние детали CTS обычно интересуют только тех, кто занимается построением инструментов и/или компиляторов, предназначенных для .NET Core. Однако всем программистам .NET Core важно знать о том, как работать с пятью типами, определенными в CTS, на выбранных ими языках. Ниже приведен краткий обзор.

Типы классов CTS

В каждом языке .NET Core поддерживается, по меньшей мере, понятие *типа класса*, которое является краеугольным камнем объектно-ориентированного программирования. Класс может состоять из любого количества членов (таких как конструкторы, свойства, методы и события) и элементов данных (полей). В языке C# классы объявляются с использованием ключевого слова `class`, примерно так:

```
// Тип класса C# с одним методом.
class Calc
{
  public int Add(int addend1, int addend2)
  {
    return addend1 + addend2;
  }
}
```

Формальное знакомство с построением типов классов в C# начнется в главе 5, а пока в таблице 1.1 приведен перечень характеристик, свойственных типам классов.

Типы интерфейсов CTS

Интерфейсы представляют собой всего лишь именованные коллекции определений и/или (начиная с версии C# 8) стандартных реализаций абстрактных членов, которые могут быть реализованными (необязательно при наличии стандартных реализаций) в заданном классе или структуре. В языке C# типы интерфейсов определяются с применением ключевого слова `interface`.

Таблица 1.1. Характеристики классов CTS

Характеристика класса	Практический смысл
Является ли класс запечатанным?	Запечатанные классы не могут выступать в качестве базовых для других классов
Реализует ли класс какие-то интерфейсы?	Интерфейс — это коллекция абстрактных членов, которая предоставляет контракт между объектом и пользователем объекта. Система CTS позволяет классу реализовывать любое количество интерфейсов
Является класс абстрактным или конкретным?	Абстрактные классы не допускают прямого создания экземпляров и предназначены для определения общего поведения производных типов. Экземпляры конкретных классов могут создаваться напрямую
Какова видимость класса?	Каждый класс должен конфигурироваться с ключевым словом видимости, таким как `public` или `internal`. По существу оно управляет тем, может ли класс использоваться во внешних сборках или же только внутри определяющей его сборки

По соглашению имена всех интерфейсов .NET Core начинаются с прописной буквы `I`, как показано в следующем примере:

```
// Тип интерфейса C# обычно объявляется как
// public, чтобы позволить типам из других
// сборок реализовывать его поведение.
public interface IDraw
{
  void Draw();
}
```

Сами по себе интерфейсы приносят не особо много пользы. Тем не менее, когда класс или структура реализует выбранный интерфейс уникальным об разом, появляется возможность получать доступ к предоставленной функциональности, используя ссылку на этот интерфейс в полиморфной манере. Программирование на основе интерфейсов подробно рассматривается в главе 8.

Типы структур CTS

Концепция *структуры* также формализована в CTS. Если вы имели дело с языком C, то вас наверняка обрадует, что эти определяемые пользователем типы (user-defined type — UDT) сохранились в мире .NET Core (хотя их внутреннее поведение несколько изменилось). Попросту говоря, *структуру* можно считать легковесным типом класса, который имеет семантику, основанную на значении. Тонкости структур более подробно исследуются в главе 4. Обычно структуры лучше всего подходят для моделирования геометрических и математических данных и создаются в языке C# с применением ключевого слова `struct`, например:

```
// Тип структуры C#.
struct Point
{
  // Структуры могут содержать поля.
  public int xPos, yPos;
```

```
  // Структуры могут содержать параметризованные конструкторы.
  public Point(int x, int y)
  { xPos = x; yPos = y;}
  // В структурах могут определяться методы.
  public void PrintPosition()
  {
    Console.WriteLine("({0}, {1})", xPos, yPos);
  }
}
```

Типы перечислений CTS

Перечисления — это удобная программная конструкция, которая позволяет группировать пары "имя-значение". Например, предположим, что требуется создать игровое приложение, в котором игроку разрешено выбирать персонажа из трех категорий: `Wizard` (маг), `Fighter` (воин) или `Thief` (вор). Вместо отслеживания простых числовых значений, представляющих каждую категорию, можно было бы создать строго типизированное перечисление, используя ключевое слово `enum`:

```
// Тип перечисления C#.
enum CharacterType
{
  Wizard = 100,
  Fighter = 200,
  Thief = 300
}
```

По умолчанию для хранения каждого элемента выделяется блок памяти, соответствующий 32-битному целому, однако при необходимости (скажем, при программировании для устройств с малым объемом памяти наподобие мобильных устройств) область хранения можно изменить. Кроме того, спецификация CTS требует, чтобы перечислимые типы были производными от общего базового класса `System.Enum`. Как будет показано в главе 4, в этом базовом классе определено несколько интересных членов, которые позволяют извлекать, манипулировать и преобразовывать лежащие в основе пары "имя-значение" программным образом.

Типы делегатов CTS

Делегаты являются эквивалентом .NET Core указателей на функции в стиле C, безопасных в отношении типов. Основная разница в том, что делегат .NET Core представляет собой *класс*, производный от `System.MulticastDelegate`, а не простой указатель на низкоуровневый адрес в памяти. В языке C# делегаты объявляются с помощью ключевого слова `delegate`:

```
// Этот тип делегата C# может "указывать" на любой метод,
// возвращающий тип int и принимающий два значения int.
delegate int BinaryOp(int x, int y);
```

Делегаты критически важны, когда объект необходимо наделить возможностью перенаправления вызова другому объекту, и они формируют основу архитектуры событий .NET Core. Как будет показано в главах 12 и 14, делегаты обладают внутренней поддержкой группового вызова (т.е. перенаправления запроса множеству получателей) и асинхронного вызова методов (т.е. вызова методов во вторичном потоке).

Члены типов CTS

Теперь, когда было представлено краткое описание каждого типа, формализованного в CTS, следует осознать тот факт, что большинство таких типов располагает любым количеством *членов*. Формально член типа ограничен набором {конструктор, финализатор, статический конструктор, вложенный тип, операция, метод, свойство, индексатор, поле, поле только для чтения, константа, событие}.

В спецификации CTS определены разнообразные *характеристики*, которые могут быть ассоциированы с заданным членом. Например, каждый член может иметь характеристику видимости (открытый, закрытый или защищенный). Некоторые члены могут быть объявлены как абстрактные (чтобы обеспечить полиморфное поведение в производных типах) или как виртуальные (чтобы определить заготовленную, но допускающую переопределение реализацию). Вдобавок большинство членов могут быть сконфигурированы как статические (связанные с уровнем класса) или члены экземпляра (связанные с уровнем объекта). Создание членов типов будет описано в нескольких последующих главах.

На заметку! Как объясняется в главе 10, в языке C# также поддерживается создание обобщенных типов и обобщенных членов.

Встроенные типы данных CTS

Финальный аспект спецификации CTS, о котором следует знать на текущий момент, заключается в том, что она устанавливает четко определенный набор фундаментальных типов данных. Хотя в каждом отдельном языке для объявления фундаментального типа данных обычно имеется уникальное ключевое слово, ключевые слова всех языков .NET Core в конечном итоге распознаются как один и тот же тип CTS, определенный в сборке по имени `mscorlib.dll`. В табл. 1.2 показано, каким образом основные типы данных CTS выражаются в языках VB.NET и C#.

Таблица 1.2. Встроенные типы данных CTS

Тип данных CTS	Ключевое слово VB	Ключевое слово C#
`System.Byte`	`Byte`	`byte`
`System.SByte`	`SByte`	`sbyte`
`System.Int16`	`Short`	`short`
`System.Int32`	`Integer`	`int`
`System.Int64`	`Long`	`long`
`System.UInt16`	`UShort`	`ushort`
`System.UInt32`	`UInteger`	`uint`
`System.UInt64`	`ULong`	`ulong`
`System.Single`	`Single`	`float`
`System.Double`	`Double`	`double`
`System.Object`	`Object`	`object`
`System.Char`	`Char`	`char`
`System.String`	`String`	`string`
`System.Decimal`	`Decimal`	`decimal`
`System.Boolean`	`Boolean`	`bool`

Учитывая, что уникальные ключевые слова в управляемом языке являются просто сокращенными обозначениями для реальных типов в пространстве имен System, больше не нужно беспокоиться об условиях переполнения/потери значимости для числовых данных или о том, как строки и булевские значения внутренне представляются в разных языках. Взгляните на следующие фрагменты кода, в которых определяются 32-битные целочисленные переменные в C# и Visual Basic с применением ключевых слов языка, а также формального типа данных CTS:

```
// Определение целочисленных переменных в C#.
int i = 0;
System.Int32 j = 0;
```

```
' Определение целочисленных переменных в VB.
Dim i As Integer = 0
Dim j As System.Int32 = 0
```

Понятие общеязыковой спецификации

Как вы уже знаете, разные языки программирования выражают одни и те же программные конструкции с помощью уникальных и специфичных для конкретного языка терминов. Например, в C# конкатенация строк обозначается с использованием операции "плюс" (+), а в VB для этого обычно применяется амперсанд (&). Даже если два разных языка выражают одну и ту же программную идиому (скажем, функцию, не возвращающую значение), то высока вероятность того, что синтаксис на первый взгляд будет выглядеть не сильно отличающимся:

```
// Ничего не возвращающий метод C#.
public void MyMethod()
{
  // Некоторый код...
}
```

```
' Ничего не возвращающий метод VB.
Public Sub MyMethod()
  ' Некоторый код...
End Sub
```

Ранее вы уже видели, что такие небольшие синтаксические вариации для исполняющей среды .NET Core несущественны, учитывая, что соответствующие компиляторы (в данном случае csc.exe и vbc.exe) выпускают похожий набор инструкций CIL. Тем не менее, языки могут также отличаться в отношении общего уровня функциональности. Например, язык .NET Core может иметь или не иметь ключевое слово для представления данных без знака и поддерживать или не поддерживать типы указателей. При таких возможных вариациях было бы идеально располагать опорными требованиями, которым удовлетворяли бы все языки .NET Core.

Спецификация CLS — это набор правил, подробно описывающих минимальное и полное множество характеристик, которые отдельный компилятор .NET Core должен поддерживать, чтобы генерировать код, обслуживаемый средой CLR и в то же время доступный в унифицированной манере всем ориентированным на платформу .NET Core языкам. Во многих отношениях CLS можно рассматривать как *подмножество* полной функциональности, определенной в CTS.

В конечном итоге CLS является набором правил, которых должны придерживаться создатели компиляторов, если они намерены обеспечить гладкое функционирование

своих продуктов в мире .NET Core. Каждое правило имеет простое название (например, "Правило номер 6"), и каждое правило описывает воздействие на тех, кто строит компиляторы, и на тех, кто (каким-либо образом) взаимодействует с ними. Самым важным в CLS является правило номер 1.

- **Правило номер 1.** Правила CLS применяются только к тем частям типа, которые видны извне определяющей сборки.

Из данного правила можно сделать корректный вывод о том, что остальные правила CLS не применяются к логике, используемой для построения внутренних рабочих деталей типа .NET Core. Единственными аспектами типа, которые должны быть согласованы с CLS, являются сами определения членов (т.е. соглашения об именовании, параметры и возвращаемые типы). В рамках логики реализации члена может применяться любое количество приемов, не соответствующих CLS, т.к. для внешнего мира это не играет никакой роли.

В целях иллюстрации ниже представлен метод `Add()` в C#, который не совместим с CLS, поскольку его параметры и возвращаемое значение используют данные без знака (что не является требованием CLS):

```
class Calc
{
  // Открытые для доступа данные без знака не совместимы с CLS!
  public ulong Add(ulong addend1, ulong addend2)
  {
    return addend1 + addend2;
  }
}
```

Тем не менее, взгляните на следующий класс, который взаимодействует с данными без знака внутри метода:

```
class Calc
{
  public int Add(int addend1, int addend2)
  {
    // Поскольку эта переменная ulong используется только
    // внутренне, совместимость с CLS сохраняется.
    ulong temp = 0;
    ...
    return addend1 + addend2;
  }
}
```

Класс `Calc` по-прежнему соблюдает правила CLS и можно иметь уверенность в том, что все языки .NET Core смогут вызывать его метод `Add()`.

Разумеется, помимо "Правила номер 1" в спецификации CLS определено множество других правил. Например, в CLS описано, каким образом заданный язык должен представлять текстовые строки, как внутренне представлять перечисления (базовый тип, применяемый для хранения их значений), каким образом определять статические члены и т.д. К счастью, для того, чтобы стать умелым разработчиком приложений .NET Core, запоминать все правила вовсе не обязательно. В общем и целом глубоко разбираться в спецификациях CTS и CLS обычно должны только создатели инструментов и компиляторов.

Обеспечение совместимости с CLS

Как вы увидите при чтении книги, в языке C# определено несколько программных конструкций, несовместимых с CLS. Однако хорошая новость заключается в том, что компилятор C# можно инструктировать о необходимости проверки кода на предмет совместимости с CLS, используя единственный атрибут .NET Core:

```
// Сообщить компилятору C# о том, что он должен
// осуществлять проверку на совместимость с CLS.
[assembly: CLSCompliant(true)]
```

Детали программирования на основе атрибутов подробно рассматриваются в главе 17. А пока следует просто запомнить, что атрибут `[CLSCompliant]` заставляет компилятор C# проверять каждую строку кода на соответствие правилам CLS. В случае обнаружения любых нарушений спецификации CLS компилятор выдаст предупреждение с описанием проблемного кода.

Понятие .NET Core Runtime

В дополнение к спецификациям CTS и CLS осталось рассмотреть финальный фрагмент головоломки — .NET Core Runtime или просто .NET Runtime. В рамках программирования термин *исполняющая среда* можно понимать как коллекцию служб, которые требуются для выполнения скомпилированной единицы кода. Например, когда разработчики на Java развертывают программное обеспечение на новом компьютере, им необходимо удостовериться в том, что на компьютере установлена виртуальная машина Java (Java Virtual Machine — JVM), которая обеспечит выполнение их программного обеспечения.

Инфраструктура .NET Core предлагает еще одну исполняющую среду. Основное отличие исполняющей среды .NET Core от упомянутых выше сред заключается в том, что исполняющая среда .NET Core обеспечивает единый четко определенный уровень выполнения, который разделяется *всеми* языками и платформами, ориентированными на .NET Core.

Различия между сборкой, пространством имен и типом

Любой из нас понимает важность библиотек кода. Главное назначение библиотек платформы — предоставлять разработчикам четко определенный набор готового кода, который можно задействовать в создаваемых приложениях. Однако C# не поставляется с какой-то специфичной для языка библиотекой кода. Взамен разработчики на C# используют нейтральные к языкам библиотеки .NET Core. Для поддержания всех типов внутри библиотек базовых классов в организованном виде внутри .NET Core широко применяется концепция *пространств имен*.

Пространство имен — это группа семантически родственных типов, которые содержатся в одной или нескольких связанных друг с другом сборках. Например, пространство имен `System.IO` содержит типы, относящиеся к файловому вводу-выводу, пространство имен `System.Data` — типы для работы с базами данных и т.д. Важно понимать, что одна сборка может содержать любое количество пространств имен, каждое из которых может иметь любое число типов.

Основное отличие между таким подходом и специфичной для языка библиотекой заключается в том, что любой язык, ориентированный на исполняющую среду .NET Core, использует *те же самые* пространства имен и *те же самые* типы. Например, следующие две программы представляют собой вездесущее приложение "Hello World", написанное на языках C# и VB:

```
// Приложение "Hello World" на языке C#.
using System;

public class MyApp
{
  static void Main()
  {
    Console.WriteLine("Hi from C#");
  }
}
```

```
' Приложение "Hello World" на языке VB.
Imports System
Public Module MyApp
  Sub Main()
    Console.WriteLine("Hi from VB")
  End Sub
End Module
```

Обратите внимание, что во всех языках применяется класс `Console`, определенный в пространстве имен `System`. Помимо очевидных синтаксических различий представленные приложения выглядят довольно похожими как физически, так и логически.

Понятно, что после освоения выбранного языка программирования для .NET Core вашей следующей целью как разработчика будет освоение изобилия типов, определенных в многочисленных пространствах имен .NET Core. Наиболее фундаментальное пространство имен, с которого нужно начать, называется `System`. Оно предлагает основной набор типов, которые вам как разработчику в .NET Core придется задействовать неоднократно. Фактически без добавления, по крайней мере, ссылки на пространство имен `System` построить сколько-нибудь функциональное приложение C# невозможно, т.к. в `System` определены основные типы данных (например, `System.Int32` и `System.String`). В табл. 1.3 приведены краткие описания некоторых (конечно же, не всех) пространств имен .NET Core, сгруппированные по функциональности.

Доступ к пространству имен программным образом

Полезно снова повторить, что пространство имен — всего лишь удобный способ логической организации связанных типов, содействующий их пониманию. Давайте еще раз обратимся к пространству имен `System`. С точки зрения разработчика можно предположить, что конструкция `System.Console` представляет класс по имени `Console`, который содержится внутри пространства имен под названием `System`. Однако с точки зрения исполняющей среды .NET Core это не так. Исполняющая среда видит только одиночный класс по имени `System.Console`.

Таблица 1.3. Избранные пространства имен .NET Core

Пространство имен	Описание
`System`	Внутри пространства имен `System` содержатся многочисленные полезные типы, которые предназначены для работы с внутренними данными, математическими вычислениями, генерацией случайных чисел, переменными среды и сборкой мусора, а также ряд распространенных исключений и атрибутов
`System.Collections` `System.Collections.Generic`	В этих пространствах имен определен набор контейнерных типов, а также базовые типы и интерфейсы, которые позволяют строить настраиваемые коллекции

`System.Data`
`System.Data.Common`
`System.Data.SqlClient`
	Эти пространства имен используются для взаимодействия с базами данных через ADO.NET

`System.IO`
`System.IO.Compression`
`System.IO.Ports`
	В этих пространствах имен определены многочисленные типы, предназначенные для работы с файловым вводом-выводом, сжатием данных и портами

`System.Reflection` `System.Reflection.Emit`	В этих пространствах имен определены типы, которые поддерживают обнаружение типов во время выполнения, а также динамическое создание типов
`System.Runtime.InteropServices`	Это пространство имен предоставляет средства, позволяющие типам .NET Core взаимодействовать с неуправляемым кодом (например, DLL-библиотеками на основе C и серверами COM) и наоборот
`System.Drawing` `System.Windows.Forms`	В этих пространствах имен определены типы, применяемые при построении настольных приложений с использованием первоначального инструментального набора .NET для создания пользовательских интерфейсов (Windows Forms)

`System.Windows`
`System.Windows.Controls`
`System.Windows.Shapes`
	Пространство имен `System.Windows` является корневым для нескольких пространств имен, которые применяются в приложениях WPF

`System.Windows.Forms` `System.Drawing`	Пространство имен `System.Windows.Forms` является корневым для нескольких пространств имен, которые используются в приложениях Windows Forms
`System.Linq` `System.Linq.Expressions`	В этих пространствах имен определены типы, применяемые при программировании с использованием API-интерфейса LINQ
`System.AspNetCore`	Это одно из многих пространств имен, которые позволяют строить веб-приложения ASP.NET Core и веб-службы REST

Окончание табл. 1.3

Пространство имен	Описание
`System.Threading` `System.Threading.Tasks`	В этих пространствах имен определены многочисленные типы для построения многопоточных приложений, которые способны распределять рабочую нагрузку по нескольким процессорам
`System.Security`	Безопасность является неотъемлемым аспектом мира .NET Core. В пространствах имен, связанных с безопасностью, содержится множество типов, которые позволяют работать с разрешениями, криптографией и т.д.
`System.Xml`	В пространствах имен, относящихся к XML, определены многочисленные типы, применяемые для взаимодействия с данными XML

В языке C# ключевое слово `using` упрощает процесс ссылки на типы, определенные в отдельном пространстве имен. Давайте посмотрим, каким образом оно работает. В приведенном ранее примере `Calc` в начале файла находится единственный оператор `using`:

```
using System;
```

Он делает возможной следующую строку кода:

```
Console.WriteLine("10 + 84 is {0}.", ans);
```

Без оператора `using` пришлось бы записывать так:

```
System.Console.WriteLine("10 + 84 is {0}.", ans);
```

Хотя определение типа с использованием полностью заданного имени позволяет делать код более читабельным, трудно не согласиться с тем, что применение ключевого слова `using` в C# значительно сокращает объем набора на клавиатуре. В настоящей книге полностью заданные имена в основном использоваться не будут (разве что для устранения установленной неоднозначности), а предпочтение отдается упрощенному подходу с применением ключевого слова `using`.

Однако не забывайте о том, что ключевое слово `using` — просто сокращенный способ указать полностью заданное имя типа. Любой из подходов дает в результате тот же самый код CIL (учитывая, что в коде CIL всегда используются полностью заданные имена) и не влияет ни на производительность, ни на размер сборки.

Ссылка на внешние сборки

В предшествующих версиях .NET Framework для установки библиотек инфраструктуры применялось общее местоположение, известное как *глобальный кеш сборок* (Global Assembly Cache — GAC). Инфраструктура .NET Core не использует GAC. Взамен каждая версия (включая младшие выпуски) устанавливается в собственное местоположение на компьютере (согласно версии). В среде Windows каждая версия исполняющей среды и SDK устанавливаются в `c:\Program Files\dotnet`.

В *большинстве* проектов .NET Core сборки добавляются путем добавления пакетов NuGet (раскрываются позже в книге). Тем не менее, приложения .NET Core, нацелен-

ные и разрабатываемые в среде Windows, по-прежнему располагают доступом к библиотекам COM, что тоже рассматривается позже в книге.

Чтобы сборка имела доступ к другой сборке, которую строите вы (или кто-то другой), необходимо добавить ссылку из вашей сборки на другую сборку и обладать физическим доступом к этой другой сборке. В зависимости от инструмента разработки, применяемого для построения приложений .NET Core, вам будут доступны различные способы информирования компилятора о том, какие сборки должны включаться в цикл компиляции.

Исследование сборки с помощью `ildasm.exe`

Если вас начинает беспокоить мысль о необходимости освоения всех пространств имен .NET Core, то просто вспомните о том, что уникальность пространству имен придает факт наличия в нем типов, которые каким-то образом *семантически связаны*. Следовательно, если в качестве пользовательского интерфейса достаточно простого консольного режима, то можно вообще не думать о пространствах имен, предназначенных для построения интерфейсов настольных и веб-приложений. Если вы создаете приложение для рисования, тогда вам вряд ли понадобятся пространства имен, ориентированные на работу с базами данных. Со временем вы изучите те пространства имен, которые больше всего соответствуют вашим потребностям в программировании.

Утилита `ildasm.exe` (Intermediate Language Disassembler — дизассемблер промежуточного языка) дает возможность загрузить любую сборку .NET Core и изучить ее содержимое, включая ассоциированный с ней манифест, код CIL и метаданные типов. Инструмент `ildasm.exe` позволяет программистам более подробно разобраться, как их код C# отображается на код CIL, и в итоге помогает понять внутреннюю механику функционирования .NET Core. Хотя для того, чтобы стать опытным программистом приложений .NET Core, использовать `ildasm.exe` вовсе *не обязательно*, настоятельно рекомендуется время от времени применять данный инструмент, чтобы лучше понимать, каким образом написанный код C# укладывается в концепции исполняющей среды.

На заметку! Утилита `ildasm.exe` не поставляется с исполняющей средой .NET 5. Получить этот инструмент в свое распоряжение можно двумя способами. Первый способ предусматривает его компиляцию из исходного кода исполняющей среды .NET 5, который доступен по ссылке `https://github.com/dotnet/runtime`. Второй и более простой способ — получить пакет Nuget по ссылке `https://www.nuget.org/packages/Microsoft.NETCore.ILDAsm/`. Удостоверьтесь в том, что выбираете корректную версию (для книги понадобится версия 5.0.0 или выше). Добавьте пакет `ILdasm` в свой проект с помощью команды `dotnet add package Microsoft.NETCore.ILDAsm --version 5.0.0`. На самом деле команда не загружает `ILDasm.exe` в ваш проект, а помещает его в папку пакета (на компьютере Windows): `%userprofile%\.nuget\packages\microsoft.netcore.ildasm\5.0.0\runtimes\native\`. Утилита `ILDasm.exe` версии 5.0.0 также включена в папку `Chapter_01` (и в папки для других глав, где применяется `ILDasm.exe`) хранилища GitHub для данной книги.

После загрузки утилиты `ildasm.exe` на свой компьютер вы можете запустить ее из командной строки и просмотреть справочную информацию. Чтобы извлечь код CIL, понадобится указать как минимум имя сборки,.

Вот пример команды:

```
ildasm /all /METADATA /out=csharp.il calc.cs.dll
```

Команда создаст файл по имени `csharp.il` и экспортирует в него все доступные данные.

Резюме

Задачей настоящей главы было формирование концептуальной основы, требуемой для освоения остального материала книги. Сначала исследовались ограничения и сложности, присущие технологиям, которые предшествовали инфраструктуре .NET Core, после чего в общих чертах было показано, как .NET Core и C# пытаются упростить текущее положение дел.

По существу .NET Core сводится к механизму исполняющей среды (.NET Runtime) и библиотекам базовых классов. Исполняющая среда способна обслуживать любые двоичные модули .NET Core (называемые сборками), которые следуют правилам управляемого кода. Вы видели, что сборки содержат инструкции CIL (в дополнение к метаданным типов и манифестам сборок), которые с помощью JIT-компилятора транслируются в инструкции, специфичные для платформы. Кроме того, вы ознакомились с ролью общеязыковой спецификации (CLS) и общей системы типов (CTS).

В следующей главе будет предложен обзор распространенных IDE-сред, которые можно применять при построении программных проектов на языке C#. Вас наверняка обрадует тот факт, что в книге будут использоваться полностью бесплатные (и богатые возможностями) IDE-среды, поэтому вы начнете изучение мира .NET Core без каких-либо финансовых затрат.

ГЛАВА 2

Создание приложений на языке C#

Как программист на языке C#, вы можете выбрать подходящий инструмент среди многочисленных средств для построения приложений .NET Core. Выбор инструмента (или инструментов) будет осуществляться главным образом на основе трех факторов: сопутствующие финансовые затраты, операционная система (ОС), используемая при разработке программного обеспечения, и вычислительные платформы, на которые оно ориентируется. Цель настоящей главы — предложить сведения об установке .NET 5 SDK и исполняющей среды, а также кратко представить флагманские IDE-среды производства Microsoft — Visual Studio Code и Visual Studio.

Сначала в главе раскрывается установка на ваш компьютер .NET 5 SDK и исполняющей среды. Затем будет исследоваться построение первого приложения на C# с помощью Visual Studio Code и Visual Studio Community Edition.

На заметку! Экранные снимки в этой и последующих главах сделаны в IDE-среде Visual Studio Code v1.51.1 или Visual Studio 2019 Community Edition v16.8.1 на компьютере с ОС Windows. Если вы хотите строить свои приложения на компьютере с другой ОС или IDE-средой, то глава укажет правильное направление, но окна выбранной вами IDE-среды будут отличаться от изображенных на экранных снимках, приводимых в тексте.

Установка .NET 5

Чтобы приступить к разработке приложений с помощью C# 9 и .NET 5 (в среде Windows, macOS или Linux), необходимо установить комплект .NET 5 SDK (который также устанавливает исполняющую среду .NET 5). Все установочные файлы для .NET и .NET Core расположены на удобном веб-сайте `www.dot.net`. Находясь на домашней странице, щелкните на кнопке Download (Загрузить) и затем на ссылке All .NET downloads (Все загрузочные файлы .NET) под заголовком .NET. После щелчка на ссылке All .NET downloads вы увидите две LTS-версии .NET Core (2.1 и 3.1) и ссылку на .NET 5.0. Щелкните на ссылке .NET 5.0 (recommended) (.NET 5.0 (рекомендуется)). На появившейся странице выберите комплект .NET 5 SDK, который подходит для вашей ОС. В примерах книги предполагается, что вы установите SDK для .NET Core версии 5.0.100 или выше, что также приведет к установке исполняющих сред .NET, ASP.NET и .NET Desktop (в Windows).

На заметку! С выходом .NET 5 станица загрузки изменилась. Теперь на ней есть три колонки с заголовками .NET, .NET Core и .NET Framework. Щелчок на ссылке All .NET Core downloads под заголовком .NET или .NET Core приводит к переходу на одну и ту же страницу. При установке Visual Studio 2019 также устанавливается .NET Core SDK и исполняющая среда.

Понятие схемы нумерации версий .NET 5

На момент написания книги актуальной версией .NET 5 SDK была 5.0.100. Первые два числа (5.0) указывают наивысшую версию исполняющей среды, на которую можно нацеливаться, в данном случае — 5.0. Это означает, что SDK также поддерживает разработку для более низких версий исполняющей среды, таких как .NET Core 3.1. Следующее число (1) представляет квартальный диапазон средств. Поскольку речь идет о первом квартале года выпуска, оно равно 1. Последние два числа (00) указывают версию исправления. Если вы добавите в уме разделитель к версии, думая о текущей версии, как о 5.0.1.00, то ситуация чуть прояснится.

Подтверждение успешности установки .NET 5

Чтобы проверить, успешно ли установлены комплект SDK и исполняющая среда, откройте окно командной подсказки и воспользуйтесь интерфейсом командной строки (CLI) .NET 5, т.е. `dotnet.exe`. В интерфейсе CLI доступны параметры и команды SDK. Команды включают создание, компиляцию, запуск и опубликование проектов и решений; позже в книге вы встретите примеры применения упомянутых команд. В этом разделе мы исследуем параметры SDK, которых четыре, как показано в табл. 2.1.

Таблица 2.1. Параметры командной строки для .NET 5 SDK

Параметр	Описание
`--version`	Отображает используемую версию .NET SDK
`--info`	Отображает информацию о .NET
`--list-runtimes`	Отображает установленные исполняющие среды
`--list-sdks`	Отображает установленные комплекты SDK

Параметр `--version` позволяет отобразить наивысшую версию комплекта SDK, установленного на компьютере, или версию, которая указана в файле `global.json`, расположенном в текущем каталоге или выше него. Проверьте версию .NET 5 SDK, установленную на компьютере, за счет ввода следующей команды:

```
dotnet --version
```

Для настоящей книги результатом должен быть 5.0.100 (или выше).

Чтобы просмотреть все исполняющие среды .NET Core, установленные на компьютере, введите такую команду:

```
dotnet --list-runtimes
```

Существует три разных исполняющих среды:

- `Microsoft.AspNetCore.App` (для построения приложений ASP.NET Core);

- `Microsoft.NETCore.App` (основная исполняющая среда для .NET Core);
- `Microsoft.WindowsDesktop.App` (для построения приложений Windows Forms и WPF).

В случае если ваш компьютер работает под управлением ОС Windows, тогда версией каждой из перечисленных исполняющих сред должна быть 5.0.0 (или выше). Для ОС, отличающихся от Windows, понадобятся первые две исполняющих среды, `Microsoft.NETCore.App` и `Microsoft.AspNetCore.App`, версией которых тоже должна быть 5.0.0 (или выше).

Наконец, чтобы увидеть все установленные комплекты SDK, введите следующую команду:

```
dotnet --list-sdks
```

И снова версией должна быть 5.0.100 (или выше).

Использование более ранних версий .NET (Core) SDK

Если вам необходимо привязать свой проект к более ранней версии .NET Core SDK, то можно воспользоваться файлом `global.json`, который создается с помощью такой команды:

```
dotnet new globaljson --sdk-version 3.1.404
```

В результате создается файл `global.json` с содержимым следующего вида:

```
{
  "sdk": {
    "version": "3.1.404"
  }
}
```

Этот файл "прикрепляет" текущий каталог и все его подкаталоги к версии 3.1.404 комплекта .NET Core SDK. Запуск команды `dotnet.exe --version` в таком каталоге возвратит 3.1.404.

Построение приложений .NET Core с помощью Visual Studio

Если у вас есть опыт построения приложений с применением технологий Microsoft предшествующих версий, то вполне вероятно, что вы знакомы с Visual Studio. На протяжении времени жизни продукта названия редакций и наборы функциональных возможностей менялись, но с момента выпуска .NET Core остались неизменными. Инструмент Visual Studio доступен в виде следующий редакций (для Window и Mac):

- Visual Studio 2019 Community (бесплатная);
- Visual Studio 2019 Professional (платная);
- Visual Studio 2019 Enterprise (платная).

Редакции Community и Professional *по существу* одинаковы. Наиболее значительная разница связана с моделью лицензирования. Редакция Community лицензируется для использования проектов с открытым кодом, в учебных учреждениях и на малых предприятиях. Редакции Professional и Enterprise являются коммерческими продуктами, которые лицензируются для любой разработки, включая корпоратив-

ную. Редакция Enterprise по сравнению с Professional вполне ожидаемо предлагает многочисленные дополнительные средства.

На заметку! Детали лицензирования доступны на веб-сайте `www.visualstudio.com`. Лицензирование продуктов Microsoft может показаться сложным и в книге его подробности не раскрываются. Для написания (и проработки) настоящей книги законно применять редакцию Community.

Все редакции Visual Studio поставляются с развитыми редакторами кода, встроенными отладчиками, конструкторами графических пользовательских интерфейсов для настольных и веб-приложений и т.д. Поскольку все они разделяют общий набор основных средств, между ними легко перемещаться и чувствовать себя вполне комфортно в отношении их стандартной эксплуатации.

Установка Visual Studio 2019 (Windows)

Чтобы продукт Visual Studio 2019 можно было использовать для разработки, запуска и отладки приложений C#, его необходимо установить. По сравнению с версией Visual Studio 2017 процесс установки значительно изменился и потому заслуживает более подробного обсуждения.

На заметку! Загрузить Visual Studio 2019 Community можно по ссылке `www.visualstudio.com/downloads`. Удостоверьтесь в том, что загружаете и устанавливаете минимум версию 16.8.1 или более позднюю.

Процесс установки Visual Studio 2019 теперь разбит на рабочие нагрузки по типам приложений. В результате появляется возможность устанавливать только те компоненты, которые нужны для построения планируемого типа приложений. Например, если вы собираетесь строить веб-приложения, тогда должны установить рабочую нагрузку ASP.NET and web development (Разработка приложений ASP.NET и веб-приложений).

Еще одно (крайне) важное изменение связано с тем, что Visual Studio 2019 поддерживает подлинную установку бок о бок. Обратите внимание, что речь идет не о параллельной установке с предшествующими версиями, а о самом продукте Visual Studio 2019! Скажем, на главном рабочем компьютере может быть установлена редакция Visual Studio 2019 Enterprise для профессиональной работы и редакция Visual Studio 2019 Community для работы с настоящей книгой. При наличии редакции Professional или Enterprise, предоставленной вашим работодателем, вы по-прежнему можете установить редакцию Community для работы над проектами с открытым кодом (или с кодом данной книги).

После запуска программы установки Visual Studio 2019 Community появляется экран, показанный на рис. 2.1. На нем предлагаются все доступные рабочие нагрузки, возможность выбора отдельных компонентов и сводка (в правой части), которая отображает, что было выбрано.

Для этой книги понадобится установить следующие рабочие нагрузки:

- .NET desktop development (Разработка классических приложений .NET)
- ASP.NET and web development (ASP.NET и разработка веб-приложений)
- Data storage and processing (Хранение и обработка данных)
- .NET Core cross-platform development (Межплатформенная разработка для .NET Core)

На вкладке Individual components (Отдельные компоненты) отметьте флажки Class Designer (Конструктор классов), Git for Windows (Git для Windows) и GitHub extension for Visual Studio (Расширение GitHub для Visual Studio) в группе Code tools (Средства для работы с кодом). После выбора всех указанных элементов щелкните на кнопке Install (Установить). В итоге вам будет предоставлено все, что необходимо для проработки примеров в настоящей книге.

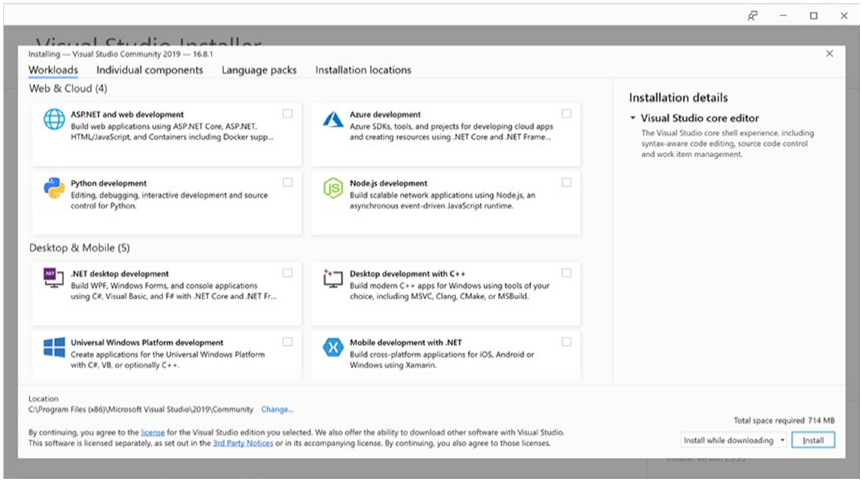

Рис. 2.1. Новая программа установки Visual Studio

Испытание Visual Studio 2019

Среда Visual Studio 2019 — это универсальный инструмент для разработки программного обеспечения с помощью платформы .NET и языка C#. Давайте бегло посмотрим на работу Visual Studio, построив простое консольное приложение .NET 5.

Использование нового диалогового окна для создания проекта и редактора кода C#

Запустив Visual Studio, вы увидите обновленное диалоговое окно запуска, которое показано на рис. 2.2. В левой части диалогового окна находятся недавно использованные решения, а в правой части — варианты запуска Visual Studio путем запуска кода из хранилища, открытия существующего проекта/решения, открытия локальной папки или создания нового проекта. Существует также вариант продолжения без кода, который обеспечивает просто запуск IDE-среды Visual Studio.

Выберите вариант Create a new project (Создать новый проект); отобразится диалоговое окно Create a new project (Создание нового проекта). Как видно на рис. 2.3, слева располагаются недавно использованные шаблоны (при их наличии), а справа — все доступные шаблоны, включая набор фильтров и поле поиска.

Начните с создания проекта типа Console App (.NET Core) (Консольное приложение (.NET Core)) на языке C#, выбрав версию C#, но не Visual Basic.

Откроется диалоговое окно Configure your new project (Конфигурирование нового проекта), представленное на рис. 2.4.

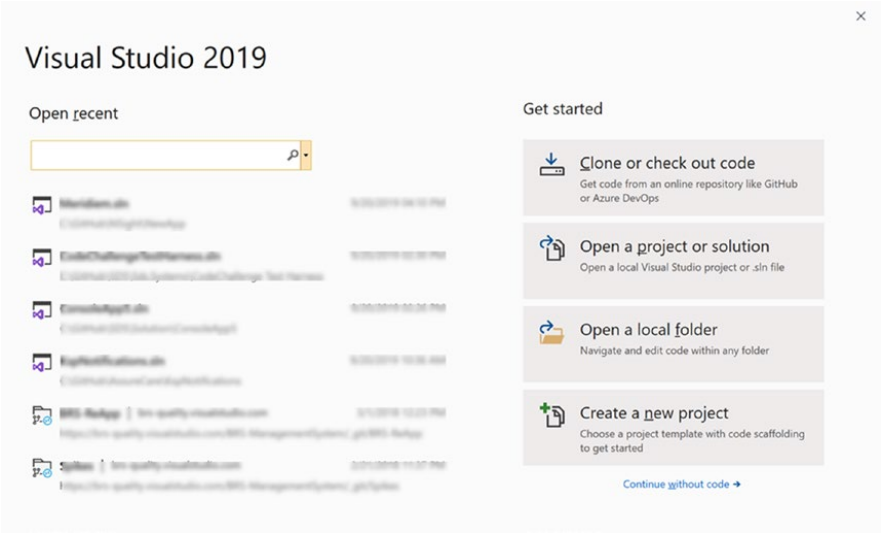

Рис. 2.2. Новое диалоговое окно запуска Visual Studio

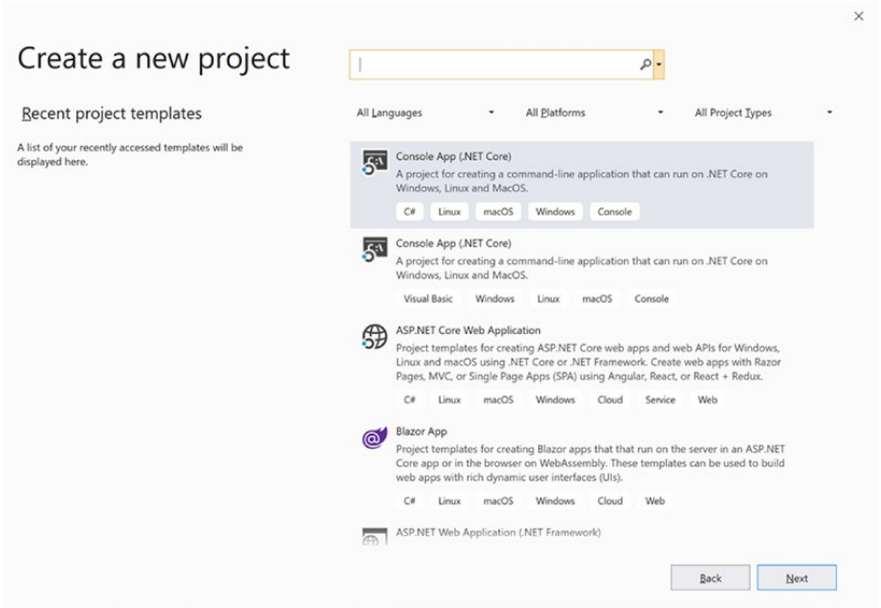

Рис. 2.3. Диалоговое окно Create a new project

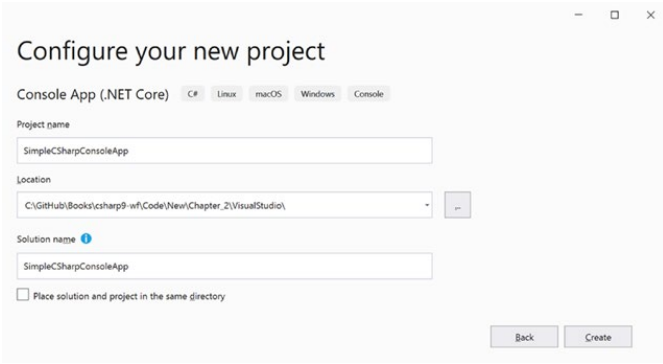

Рис. 2.4. Диалоговое окно Configure your new project

Введите `SimpleCSharpConsoleApp` в качестве имени проекта и выберите местоположение для проекта. Мастер также создаст решение Visual Studio, по умолчанию получающее имя проекта.

На заметку! Создавать решения и проекты можно также с применением интерфейса командной строки .NET Core, как будет объясняться при рассмотрении Visual Studio Code.

После создания проекта вы увидите начальное содержимое файла кода C# (по имени `Program.cs`), который открывается в редакторе кода. Замените единственную строку кода в методе `Main()` приведенным ниже кодом. По мере набора кода вы заметите, что во время применения операции точки активизируется средство IntelliSense.

```
static void Main(string[] args)
{
  // Настройка консольного пользовательского интерфейса.
  Console.Title = "My Rocking App";
  Console.ForegroundColor = ConsoleColor.Yellow;
  Console.BackgroundColor = ConsoleColor.Blue;
  Console.WriteLine("*************************************");
  Console.WriteLine("***** Welcome to My Rocking App *****");
  Console.WriteLine("*************************************");
  Console.BackgroundColor = ConsoleColor.Black;

  // Ожидание нажатия клавиши <Enter>.
  Console.ReadLine();
}
```

Здесь используется класс `Console`, определенный в пространстве имен `System`. Поскольку пространство имен `System` было автоматически включено посредством оператора `using` в начале файла, указывать `System` перед именем класса не обязательно (например, `System.Console.WriteLine()`). Данная программа не делает ничего особо интересного; тем не менее, обратите внимание на последний вызов `Console.ReadLine()`. Он просто обеспечивает поведение, при котором пользователь должен нажать клавишу <Enter>, чтобы завершить приложение. При работе в Visual Studio 2019 поступать так не обязательно, потому что встроенный отладчик приостановит про-

грамму, предотвращая ее завершение. Но без вызова `Console.ReadLine()` при запуске скомпилированной версии программа прекратит работу почти мгновенно!

На заметку! Если вы хотите, чтобы отладчик VS завершал программу автоматически, тогда установите флажок Automatically close the console when debugging stops (Автоматически закрывать окно консоли при останове отладки) в диалоговом окне, доступном через пункт меню Tools⇨Options⇨Debugging (Сервис⇨Параметры⇨Отладка).

Изменение целевой инфраструктуры .NET Core

Стандартной версией .NET Core для консольных приложений .NET Core и библиотек классов является последняя версия LTS — .NET Core 3.1. Чтобы использовать .NET 5 или просто проверить версию .NET (Core), на которую нацелено ваше приложение, дважды щелкните на имени проекта в окне Solution Explorer (Проводник решений). В окне редактора откроется файл проекта (новая возможность Visual Studio 2019 и .NET Core). Отредактировать файл проекта можно также, щелкнув правой кнопкой мыши на имени проекта в окне Solution Explorer и выбрав в контекстном меню пункт Edit Project file (Редактировать файл проекта). Вы увидите следующее содержимое:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
</Project>
```

Для смены версии .NET Core на .NET 5 измените значение `TargetFramework` на `net5.0`:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Изменить целевую инфраструктуру можно и по-другому. Щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer, выберите в контекстном меню пункт Properties (Свойства), в открывшемся диалоговом окне Properties (Свойства) перейдите на вкладку Application (Приложение) и выберите в раскрывающемся списке Target framework (Целевая инфраструктура) вариант .NET 5.0, как показано на рис. 2.5.

Использование функциональных средств C# 9

В предшествующих версиях .NET можно было изменять версию C#, поддерживаемую проектом. С выходом .NET Core 3.0+ применяемая версия C# привязана к версии инфраструктуры. Чтобы удостовериться в этом, щелкните правой кнопкой на имени проекта в окне Solution Explorer и выберите в контекстном меню пункт Properties (Свойства). В открывшемся диалоговом окне Properties (Свойства) перейдите на вкладку Build (Сборка) и щелкните на кнопке Advanced (Дополнительно) в нижнем правом углу. Откроется диалоговое окно Advanced Build Settings (Расширенные настройки сборки), представленное на рис. 2.6.

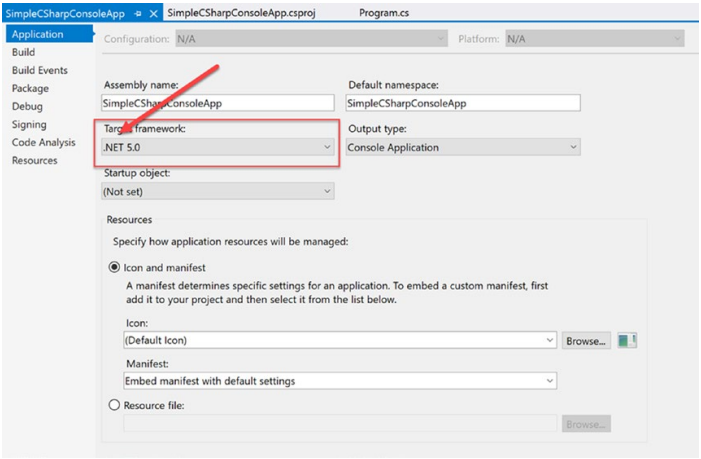

Рис. 2.5. Изменение целевой инфраструктуры для приложения

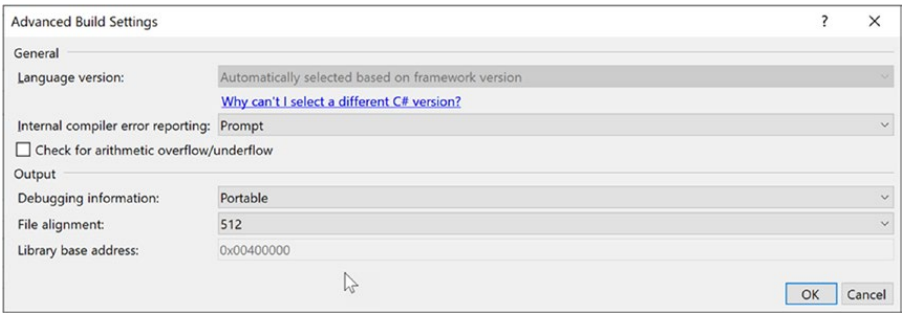

Рис. 2.6. Диалоговое окно Advanced Build Settings

Для проектов .NET 5.0 версия языка зафиксирована как C# 9. В табл. 2.2 перечислены целевые инфраструктуры (.NET Core, .NET Standard и .NET Framework) и задействованные по умолчанию версии C#.

Таблица 2.2. Версии C# и целевые инфраструктуры

Целевая инфраструктура	Версия	Версия языка C#, используемая по умолчанию
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	Все	C# 7.3

Запуск и отладка проекта

Чтобы запустить программу и просмотреть вывод, нажмите комбинацию клавиш <Ctrl+F5> (или выберите пункт меню Debug⇨Start Without Debugging (Отладка⇨Запустить без отладки)). На экране появится окно консоли Windows с вашим специальным (раскрашенным) сообщением. Имейте в виду, что при "запуске" своей программы с помощью <Ctrl+F5> вы обходите интегрированный отладчик.

На заметку! Приложения .NET Core можно компилировать и запускать также с применением интерфейса командной строки. Чтобы запустить свой проект, введите команду `dotnet run` в каталоге, где находится файл проекта (`SimpleCSharpApp.csproj` в рассматриваемом примере). Команда `dotnet run` вдобавок автоматически компилирует проект.

Если написанный код необходимо отладить (что определенно будет важным при построении более крупных программ), тогда первым делом понадобится поместить точки останова на операторы кода, которые необходимо исследовать. Хотя в рассматриваемом примере не особо много кода, установите точку останова, щелкнув на крайней слева полосе серого цвета в окне редактора кода (обратите внимание, что точки останова помечаются значком в виде красного кружка (рис. 2.7)).

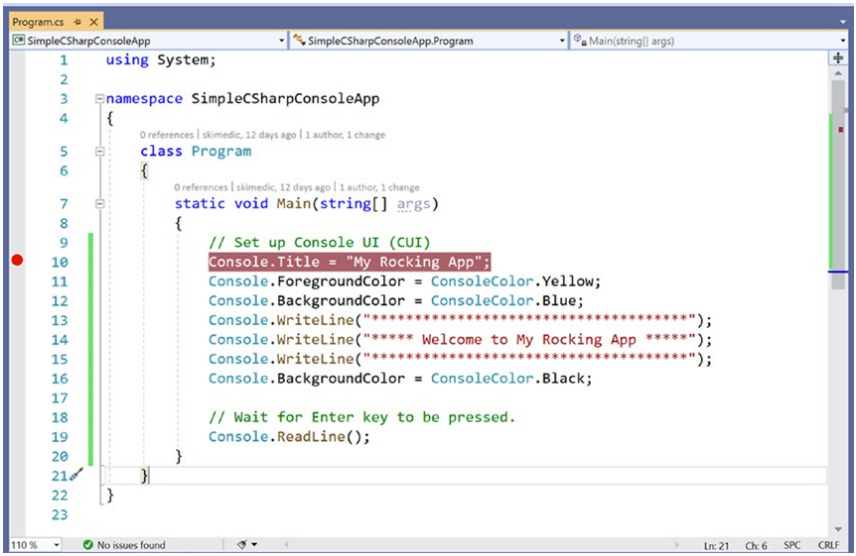

Рис. 2.7. Установка точки останова

Если теперь нажать клавишу <F5> (или выбрать пункт меню Debug⇨Start Debugging (Отладка⇨Запустить с отладкой) либо щелкнуть на кнопке с зеленой стрелкой и надписью Start (Пуск) в панели инструментов), то программа будет прекращать работу на каждой точке останова. Как и можно было ожидать, у вас есть возможность взаимодействовать с отладчиком с помощью разнообразных кнопок панели инструментов и пунктов меню IDE-среды. После прохождения всех точек останова приложение в конечном итоге завершится, когда закончится метод `Main()`.

На заметку! Предлагаемые Microsoft среды IDE снабжены современными отладчиками, и в последующих главах вы изучите разнообразные приемы работы с ними. Пока нужно лишь знать, что при нахождении в сеансе отладки в меню Debug появляется большое количество полезных пунктов. Выделите время на ознакомление с ними.

Использование окна Solution Explorer

Взглянув на правую часть IDE-среды, вы заметите окно Solution Explorer (Проводник решений), в котором отображено несколько важных элементов. Первым делом обратите внимание, что IDE-среда создала решение с единственным проектом. Поначалу это может сбивать с толку, т.к. решение и проект имеют одно и то же имя (`SimpleCSharpConsoleApp`). Идея в том, что "решение" может содержать множество проектов, работающих совместно. Скажем, в состав решения могут входить три библиотеки классов, одно приложение WPF и одна веб-служба ASP.NET Core. В начальных главах книги будет всегда применяться одиночный проект; однако, когда мы займемся построением более сложных примеров, будет показано, каким образом добавлять новые проекты в первоначальное пространство решения.

На заметку! Учтите, что в случае выбора решения в самом верху окна Solution Explorer система меню IDE-среды будет отображать набор пунктов, который отличается от ситуации, когда выбран проект. Если вы когда-нибудь обнаружите, что определенный пункт меню исчез, то проверьте, не выбран ли случайно неправильный узел.

Использование визуального конструктора классов

Среда Visual Studio также снабжает вас возможностью конструирования классов и других типов (вроде интерфейсов или делегатов) в визуальной манере. Утилита Class Designer (Визуальный конструктор классов) позволяет просматривать и модифицировать отношения между типами (классами, интерфейсами, структурами, перечислениями и делегатами) в проекте. С помощью данного средства можно визуально добавлять (или удалять) члены типа с отражением этих изменений в соответствующем файле кода C#. Кроме того, по мере модификации отдельного файла кода C# изменения отражаются в диаграмме классов.

Для доступа к инструментам визуального конструктора классов сначала понадобится вставить новый файл диаграммы классов. Выберите пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент) и в открывшемся окне найдите элемент Class Diagram (Диаграмма классов), как показано на рис. 2.8.

Первоначально поверхность визуального конструктора будет пустой; тем не менее, вы можете перетаскивать на нее файлы из окна Solution Explorer. Например, после перетаскивания на поверхность конструктора файла `Program.cs` вы увидите визуальное представление класса `Program`. Щелкая на значке с изображением стрелки для заданного типа, можно отображать или скрывать его члены (рис. 2.9).

На заметку! Используя панель инструментов утилиты Class Designer, можно настраивать параметры отображения поверхности визуального конструктора.

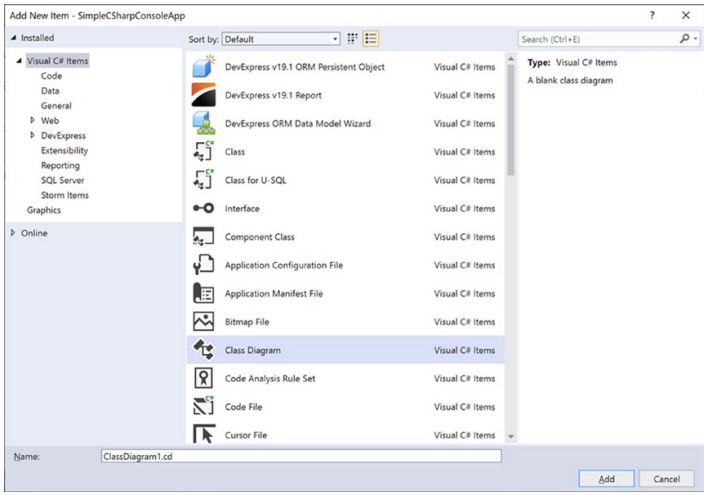

Рис. 2.8. Вставка файла диаграммы классов в текущий проект

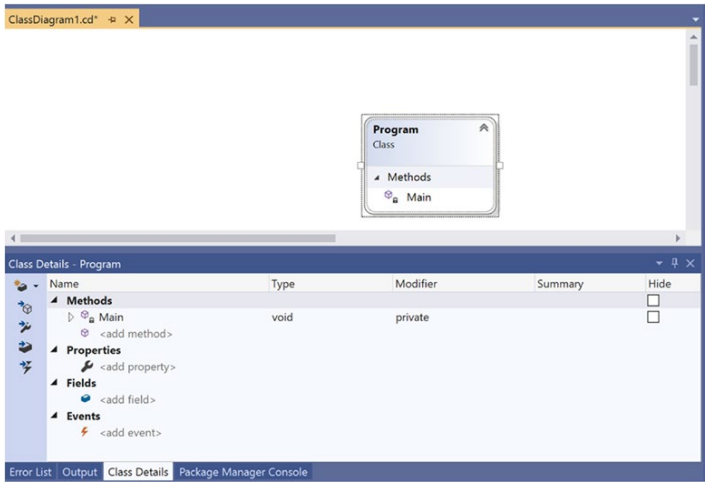

Рис. 2.9. Просмотр диаграммы классов

Утилита Class Designer работает в сочетании с двумя другими средствами Visual Studio — окном Class Details (Детали класса), которое открывается через меню View⇨Other Windows (Вид⇨Другие окна), и панелью инструментов Class Designer, отображаемой выбором пункта меню View⇨Toolbox (Вид⇨Панель инструментов). Окно Class Details не только показывает подробные сведения о текущем выбранном элементе диаграммы, но также позволяет модифицировать существующие члены и вставлять новые члены на лету (рис. 2.10).

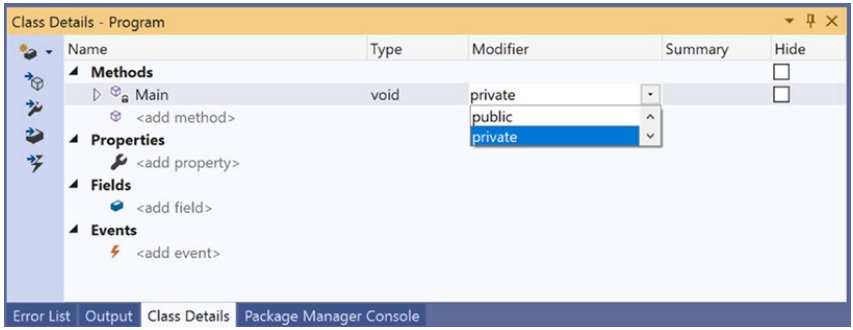

Рис. 2.10. Окно Class Details

Панель инструментов Class Designer, которая также может быть активизирована с применением меню View, позволяет вставлять в проект новые типы (и создавать между ними отношения) визуальным образом (рис. 2.11). (Чтобы видеть эту панель инструментов, должно быть активным окно диаграммы классов.) По мере выполнения таких действий IDE-среда создает на заднем плане новые определения типов на C#.

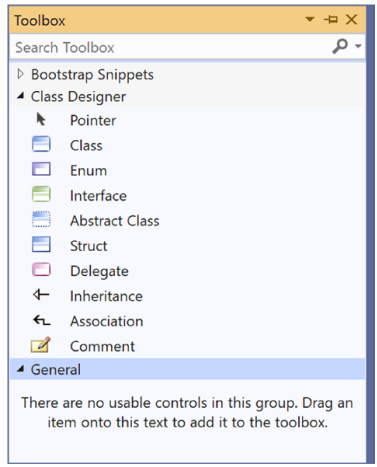

Рис. 2.11. Панель инструментов Class Designer

В качестве примера перетащите элемент Class (Класс) из панели инструментов Class Designer в окно Class Designer. В открывшемся диалоговом окне назначьте ему имя `Car`. В результате создается новый файл C# по имени `Car.cs` и автоматически добавляется к проекту. Теперь, используя окно Class Details, добавьте открытое поле типа `string` с именем `PetName` (рис. 2.12).

Заглянув в определение C# класса `Car`, вы увидите, что оно было соответствующим образом обновлено (за исключением приведенного ниже комментария):

```
public class Car
{
  // Использовать открытые данные
  // обычно не рекомендуется,
  // но здесь это упрощает пример.
  public string petName;
}
```

Снова активизируйте утилиту Class Designer, перетащите на поверхность визуального конструктора еще один элемент Class и назначьте ему имя `SportsCar`. Далее выберите значок Inheritance (Наследование) в панели инструментов Class Designer и щелкните в верхней части значка `SportsCar`. Щелкните в верхней части значка класса `Car`. Если все было сделано правильно, тогда класс `SportsCar` станет производным от класса `Car` (рис. 2.13).

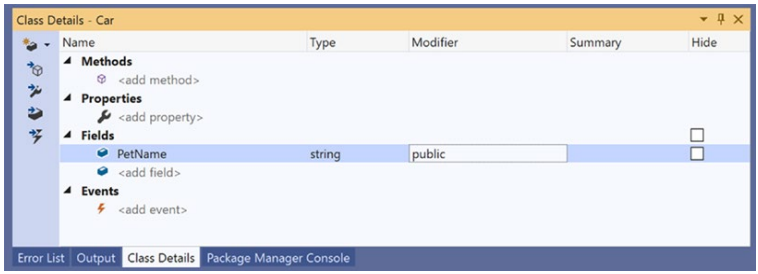

Рис. 2.12. Добавление поля с помощью окна Class Details

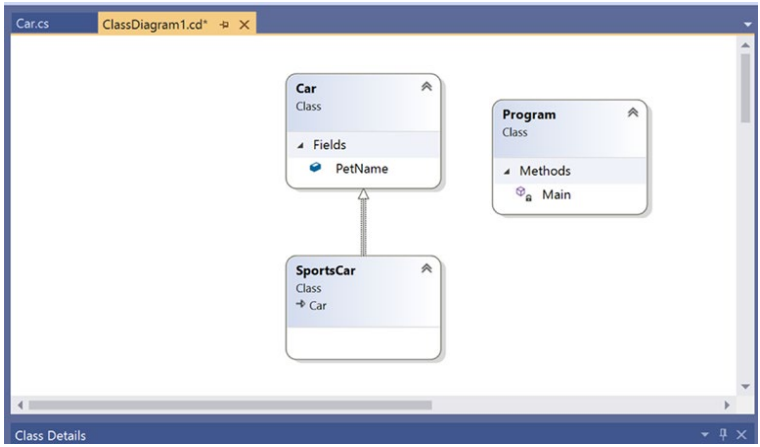

Рис. 2.13. Визуальное наследование от существующего класса

На заметку! Концепция наследования подробно объясняется в главе 6.

Чтобы завершить пример, обновите сгенерированный класс `SportsCar`, добавив открытый метод по имени `GetPetName()` со следующим кодом:

```
public class SportsCar : Car
{
  public string GetPetName()
  {
    petName = "Fred";
    return petName;
  }
}
```

Как и можно было ожидать, визуальный конструктор отобразит метод, добавленный в класс `SportsCar`.

На этом краткий обзор Visual Studio завершен. В оставшемся материале книги вы встретите дополнительные примеры применения Visual Studio для построения приложений с использованием C# 9 и .NET 5.

Построение приложений .NET Core с помощью Visual Studio Code

Еще одной популярной IDE-средой от Microsoft следует считать Visual Studio Code (VSC). Продукт VSC — относительно новая редакция в семействе Microsoft. Он является бесплатным и межплатформенным, поставляется с открытым кодом и получил широкое распространение среди разработчиков в экосистеме .NET Core и за ее пределами. Как должно быть понятно из названия, в центре внимания Visual Studio Code находится код вашего приложения. Продукт VSC лишен многих встроенных средств, входящих в состав Visual Studio. Однако существуют дополнительные средства, которые можно добавить к VSC через расширения, что позволяет получить быструю IDE-среду, настроенную для имеющегося рабочего потока. Многочисленные примеры в данной книге были собраны и протестированы с помощью VSC. Загрузить VSC можно по ссылке `https://code.visualstudio.com/download`.

После установки VSC вы наверняка захотите добавить расширение C#, которое доступно по следующей ссылке:

```
https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp
```

На заметку! Продукт VSC используется для разработки разных видов приложений на основе множества языков. Существуют расширения для Angular, View, PHP, Java и многих других языков.

Испытание Visual Studio Code

Давайте применим VSC для построения того же самого консольного приложения .NET 5, которое создавалось ранее в Visual Studio.

Создание решений и проектов

После запуска VSC вы начинаете с "чистого листа". Решения и проекты должны создаваться с использованием интерфейса командной строки (command-line interface — CLI) платформы .NET 5. Первым делом откройте папку в VSC, выбрав пункт меню File⇨Open Folder (Файл⇨Открыть папку), и с помощью окна проводника перейдите туда, куда планируется поместить решение и проект. Затем откройте терминальное окно, выбрав пункт меню Terminal⇨New Terminal (Терминал⇨Новое терминальное окно) или нажав комбинацию клавиш <Ctl+Shift+`>.

Введите в терминальном окне следующую команду, чтобы создать пустой файл решения .NET 5:

```
dotnet new sln -n SimpleCSharpConsoleApp -o .\VisualStudioCode
```

Команда создает новый файл решения с именем `SimpleCSharpConsoleApp` (указанным посредством `-n`) в подкаталоге (внутри текущего каталога) по имени `VisualStudioCode`. В случае применения VSC с единственным проектом нет необходимости в создании файла решения. Продукт Visual Studio ориентирован на решения, а Visual Studio Code — на код. Файл решения здесь создан для того, чтобы повторить процесс построения примера приложения в Visual Studio.

На заметку! В примерах используются разделители каталогов Windows. Вы должны применять разделители, принятые в вашей операционной системе.

Далее создайте новое консольное приложение C# 9/.NET 5 (`-f net5.0`) по имени `SimpleCSharpConsoleApp` (`-n`) в подкаталоге (`-o`) с таким же именем (команда должна вводиться в одной строке):

```
dotnet new console -lang c# -n SimpleCSharpConsoleApp -o .\VisualStudioCode\
SimpleCSharpConsoleApp -f net5.0
```

На заметку! Поскольку целевая инфраструктура была указана с использованием параметра `-f`, обновлять файл проекта, как делалось в Visual Studio, не понадобится.

Наконец, добавьте созданный проект к решению с применением следующей команды:

```
dotnet sln .\VisualStudioCode\SimpleCSharpConsoleApp.sln
  add .\VisualStudioCode\SimpleCSharpConsoleApp
```

На заметку! Это всего лишь небольшой пример того, на что способен интерфейс командной строки. Чтобы выяснить, что CLI может делать, введите команду `dotnet -h`.

Исследование рабочей области Visual Studio Code

Как легко заметить на рис. 2.14, рабочая область VSC ориентирована на код, но также предлагает множество дополнительных средств, предназначенных для повышения вашей продуктивности. Проводник (1) представляет собой встроенный проводник файлов и выбран на рисунке. Управление исходным кодом (2) интегрируется с Git. Значок отладки (3) отвечает за запуск соответствующего отладчика (исходя из предположения о том, что установлено корректное расширение). Ниже находится диспетчер расширений (4). Щелчок на значке отладки приводит к отображению списка рекомендуемых и всех доступных расширений. Диспетчер расширений чувствителен к контексту и будет выдавать рекомендации на основе типа кода в открытом каталоге и подкаталогах.

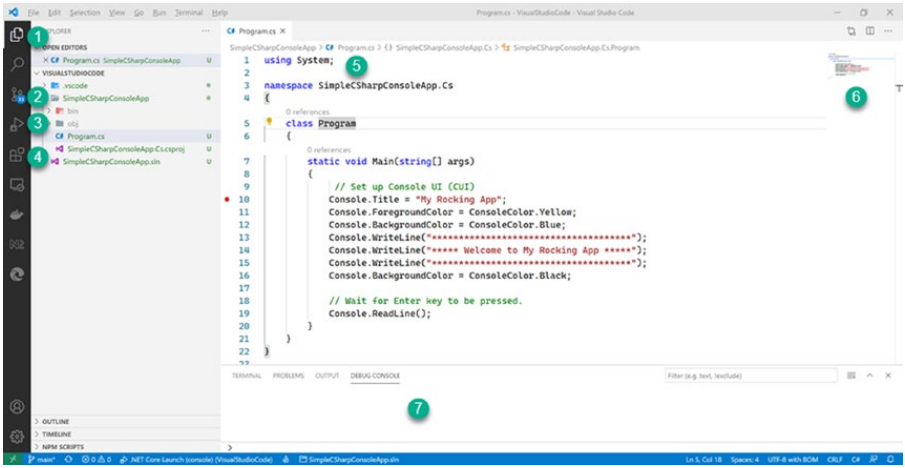

Рис. 2.14. Рабочая область VSC

Редактор кода (5) снабжен цветовым кодированием и поддержкой IntelliSense; оба средства полагаются на расширения. Кодовая карта (6) показывает карту всего файла кода, а консоль отладки (7) получает вывод из сеансов отладки и принимает ввод от пользователя (подобно окну Immediate (Интерпретация) в Visual Studio).

Восстановление пакетов, компиляция и запуск программ

Интерфейс командной строки .NET 5 обладает всеми возможностями для восстановления пакетов, сборки решений, компиляции проектов и запуска приложений. Чтобы восстановить все пакеты NuGet, требуемые для вашего решения и проекта, введите в терминальном окне (или в окне командной подсказки вне VSC) приведенную ниже команду, находясь в каталоге, который содержит файл решения:

```
dotnet restore
```

Чтобы скомпилировать все проекты в решении, введите в терминальном окне или в окне командной подсказки следующую команду (снова находясь в каталоге, где содержится файл решения):

```
dotnet build
```

На заметку! Когда команды `dotnet restore` и `dotnet build` выполняются в каталоге, содержащем файл решения, они воздействуют на все проекты в решении. Команды также можно запускать для одиночного проекта, вводя их в каталоге с файлом проекта C# (`*.csproj`).

Чтобы запустить проект без отладки, введите в каталоге с файлом проекта (`SimpleCSharpConsoleApp.csproj`) следующую команду .NET CLI:

```
dotnet run
```

Отладка проекта

Для запуска отладки проекта нажмите клавишу <F5> или щелкните на значке отладки (на рис. 2.14 она помечена цифрой 2). Исходя из предположения, что вы загрузили расширение C# для VSC, программа запустится в режиме отладки. Управление точками останова производится точно так же, как в Visual Studio, хотя в редакторе они не настолько четко выражены (рис. 2.15).

Чтобы сделать терминальное окно интегрированным и разрешить вашей программе ввод, откройте файл `launch.json` (находящийся в каталоге `.vscode`). Измените запись `"console"` с `internalConsole` на `integratedTerminal`, как показано ниже:

```
{
  // Используйте IntelliSense, чтобы выяснить, какие атрибуты
  // существуют для отладки C#.
  // Наводите курсор на существующие атрибуты, чтобы получить их описание.
  // Дополнительные сведения ищите по ссылке
  // https://github.com/OmniSharp/omnisharp-vscode/blob/master/
  // debugger-launchjson.md
  "version": "0.2.0",
  "configurations": [
    {
      "name": ".NET Core Launch (console)",
      "type": "coreclr",
```

```
      "request": "launch",
      "preLaunchTask": "build",
      // Если вы изменили целевые платформы, тогда не забудьте
      // обновить путь в program.
      "program": "${workspaceFolder}/SimpleCSharpConsoleApp/bin/
Debug/net5.0/SimpleCSharpConsoleApp.Cs.dll",
      "args": [],
      "cwd": "${workspaceFolder}/SimpleCSharpConsoleApp",
      // Дополнительные сведения об атрибуте console ищите по ссылке
      // https://code.visualstudio.com/docs/editor/
      // debugging#_launchjson-attributes
      "console": "integratedTerminal",
      "stopAtEntry": false
    },
    {
      "name": ".NET Core Attach",
      "type": "coreclr",
      "request": "attach",
      "processId": "${command:pickProcess}"
    }
  ]
}
```

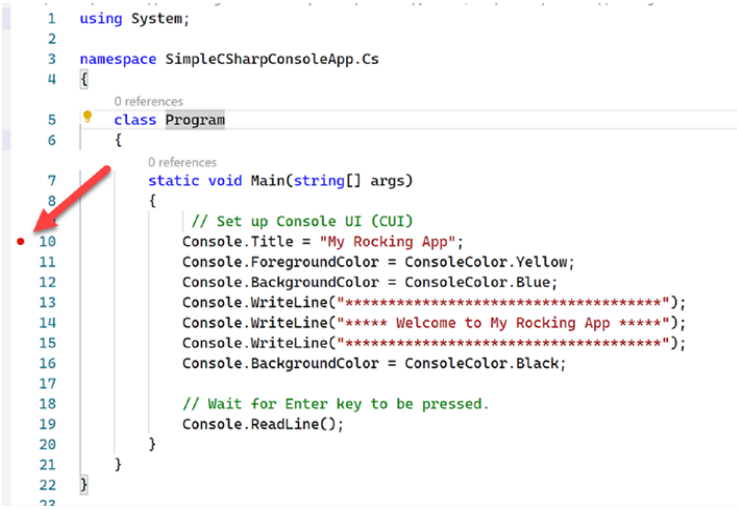

Рис. 2.15. Точки останова в VSC

Документация по .NET Core и C#

Документация .NET Core и C# представляет собой исключительно хороший, понятный и насыщенный полезной информацией источник. Учитывая огромное количество предопределенных типов .NET (их тысячи), вы должны быть готовы засучить рукава и тщательно исследовать предлагаемую документацию. Вся документация от Microsoft доступна по ссылке `https://docs.microsoft.com/ru-ru/dotnet/`.

В первой половине книги вы будете чаще всего использовать документацию по C# и документацию по .NET Core, которые доступны по следующим ссылкам:

```
https://docs.microsoft.com/ru-ru/dotnet/csharp/
https://docs.microsoft.com/ru-ru/dotnet/fundamentals/
```

Резюме

Цель этой главы заключалась в том, чтобы предоставить информацию по настройке вашей среды разработки с комплектом .NET 5 SDK и исполняющими средами, а также провести краткий экскурс в Visual Studio 2019 Community Edition и Visual Studio Code. Если вас интересует только построение межплатформенных приложений .NET Core, то доступно множество вариантов. Visual Studio (только Windows), Visual Studio для Mac (только Mac) и Visual Studio Code (межплатформенная версия) поставляются компанией Microsoft. Построение приложений WPF или Windows Forms по-прежнему требует Visual Studio на компьютере с Windows.

ЧАСТЬ II

Основы программирования на C#

ГЛАВА 3

Главные конструкции программирования на C#: часть 1

В настоящей главе начинается формальное изучение языка программирования C# за счет представления набора отдельных тем, которые необходимо знать для освоения инфраструктуры .NET Core. В первую очередь мы разберемся, каким образом строить *объект приложения*, и выясним структуру точки входа исполняемой программы, т.е. метода `Main()`, а также новое средство C# 9.0 — операторы верхнего уровня. Затем мы исследуем фундаментальные типы данных C# (и их эквиваленты в пространстве имен `System`), в том числе классы `System.String` и `System.Text.StringBuilder`.

После ознакомления с деталями фундаментальных типов данных .NET Core мы рассмотрим несколько приемов преобразования типов данных, включая сужающие и расширяющие операции, а также использование ключевых слов `checked` и `unchecked`.

Кроме того, в главе будет описана роль ключевого слова `var` языка C#, которое позволяет *неявно* определять локальную переменную. Как будет показано далее в книге, неявная типизация чрезвычайно удобна (а порой и обязательна) при работе с набором технологий LINQ. Глава завершается кратким обзором ключевых слов и операций C#, которые дают возможность управлять последовательностью выполняемых в приложении действий с применением разнообразных конструкций циклов и принятия решений.

Структура простой программы C#

Язык C# требует, чтобы вся логика программы содержалась внутри определения типа (вспомните из главы 1, что *тип* — это общий термин, относящийся к любому члену из множества {класс, интерфейс, структура, перечисление, делегат}). В отличие от многих других языков программирования создавать глобальные функции или глобальные элементы данных в C# невозможно. Взамен все данные-члены и все методы должны находиться внутри определения типа. Первым делом создадим новое пустое решение под названием `Chapter3_AllProject.sln`, которое содержит проект консольного приложения по имени `SimpleCSharpApp`.

Выберите в Visual Studio шаблон Blank Solution (Пустое решение) в диалоговом окне Create a new project (Создание нового проекта). После открытия решения щелкните правой кнопкой мыши на имени решения в окне Solution Explorer (Проводник решений)

и выберите в контекстном меню пункт Add⇨New Project (Добавить⇨Новый проект). Выберите шаблон Console App (.NET Core) (Консольное приложение (.NET Core)) на языке C#, назначьте ему имя SimpleCSharpApp и щелкните на кнопке Create (Создать). Не забудьте выбрать в раскрывающемся списке Target framework (Целевая инфраструктура) вариант .NET 5.0.

Введите в окне командной строки следующие команды:

```
dotnet new sln -n Chapter3_AllProjects
dotnet new console -lang c# -n SimpleCSharpApp
  -o .\SimpleCSharpApp -f net5.0
dotnet sln .\Chapter3_AllProjects.sln add .\SimpleCSharpApp
```

Наверняка вы согласитесь с тем, что код в первоначальном файле Program.cs не особо примечателен:

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

Теперь модифицируем метод Main() класса Program следующим образом:

```
class Program
{
  static void Main(string[] args)
  {
    // Вывести пользователю простое сообщение.
    Console.WriteLine("***** My First C# App *****");
    Console.WriteLine("Hello World!");
    Console.WriteLine();
    // Ожидать нажатия клавиши <Enter>, прежде чем завершить работу.
    Console.ReadLine();
  }
}
```

На заметку! Язык программирования C# чувствителен к регистру. Следовательно, Main — не то же, что main, а Readline — не то же, что ReadLine. Запомните, что все ключевые слова C# вводятся в нижнем регистре (например, public, lock, class, dynamic), в то время как названия пространств имен, типов и членов (по соглашению) начинаются с заглавной буквы и имеют заглавные буквы в любых содержащихся внутри словах (скажем, Console.WriteLine, System.Windows.MessageBox, System.Data.SqlClient). Как правило, каждый раз, когда вы получаете от компилятора сообщение об ошибке, касающееся неопределенных символов, то в первую очередь должны проверить правильность написания и регистр.

Предыдущий код содержит определение типа класса, который поддерживает единственный метод по имени `Main()`. По умолчанию среда Visual Studio назначает классу, определяющему метод `Main()`, имя `Program`; однако при желании его можно изменить. До выхода версии C# 9.0 каждое исполняемое приложение C# (консольная программа, настольная программа Windows или Windows-служба) должно было содержать класс, определяющий метод `Main()`, который использовался для обозначения точки входа в приложение.

Выражаясь формально, класс, в котором определен метод `Main()`, называется *объектом приложения*. В одном исполняемом приложении допускается иметь несколько объектов приложений (что может быть удобно при модульном тестировании), но тогда вы обязаны проинформировать компилятор о том, какой из методов `Main()` должен применяться в качестве точки входа. Это можно делать через элемент `<StartupObject>` в файле проекта или посредством раскрывающегося списка Startup Object (Объект запуска) на вкладке Application (Приложение) окна свойств проекта в Visual Studio.

Обратите внимание, что сигнатура метода `Main()` снабжена ключевым словом `static`, которое подробно объясняется в главе 5. Пока достаточно знать, что статические члены имеют область видимости уровня класса (а не уровня объекта) и потому могут вызываться без предварительного создания нового экземпляра класса.

Помимо наличия ключевого слова `static` метод `Main()` принимает единственный параметр, который представляет собой массив строк (`string[] args`). Несмотря на то что в текущий момент данный массив никак не обрабатывается, параметр `args` может содержать любое количество входных аргументов командной строки (доступ к ним будет вскоре описан). Наконец, метод `Main()` в примере был определен с возвращаемым значением `void`, т.е. перед выходом из области видимости метода мы не устанавливаем явным образом возвращаемое значение с использованием ключевого слова `return`.

Внутри метода `Main()` содержится логика класса `Program`. Здесь мы работаем с классом `Console`, который определен в пространстве имен `System`. В состав его членов входит статический метод `WriteLine()`, который отправляет текстовую строку и символ возврата каретки в стандартный поток вывода. Кроме того, мы производим вызов метода `Console.ReadLine()`, чтобы окно командной строки, открываемое IDE-средой Visual Studio, оставалось видимым. Когда консольные приложения .NET Core запускаются в IDE-среде Visual Studio (в режиме отладки или выпуска), то окно консоли остается видимым по умолчанию. Такое поведение можно изменить, установив флажок Automatically close the console when debugging stops (Автоматически закрывать окно консоли при останове отладки) в диалоговом окне, которое доступно через пункт меню Tools⇨Options⇨Debugging (Сервис⇨Параметры⇨Отладка). Вызов `Console.ReadLine()` здесь оставляет окно открытым, если программа выполняется из проводника Windows двойным щелчком на имени файла `*.exe`. Класс `System.Console` более подробно рассматривается далее в главе.

Использование вариаций метода `Main()` (обновление в версии 7.1)

По умолчанию Visual Studio будет генерировать метод `Main()` с возвращаемым значением `void` и одним входным параметром в виде массива строк. Тем не менее, это не единственно возможная форма метода `Main()`. Точку входа в приложение разрешено создавать с использованием любой из приведенных ниже сигнатур (предполагая, что они содержатся внутри определения класса или структуры C#):

```
// Возвращаемый тип int, массив строк в качестве параметра.
static int Main(string[] args)
{
  // Перед выходом должен возвращать значение!
  return 0;
}
// Нет возвращаемого типа, нет параметров.
static void Main()
{
}
// Возвращаемый тип int, нет параметров.
static int Main()
{
  // Перед выходом должен возвращать значение!
  return 0;
}
```

С выходом версии C# 7.1 метод `Main()` может быть асинхронным. Асинхронное программирование раскрывается в главе 15, но теперь важно помнить о существовании четырех дополнительных сигнатур:

```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

На заметку! Метод `Main()` может быть также определен как открытый в противоположность закрытому, что подразумевается, если конкретный модификатор доступа не указан. Среда Visual Studio определяет метод `Main()` как неявно закрытый. Модификаторы доступа подробно раскрываются в главе 5.

Очевидно, что выбор способа создания метода `Main()` зависит от ответов на три вопроса. Первый вопрос: нужно ли возвращать значение системе, когда метод `Main()` заканчивается и работа программы завершается? Если да, тогда необходимо возвращать тип данных `int`, а не `void`. Второй вопрос: требуется ли обрабатывать любые предоставляемые пользователем параметры командной строки? Если да, то они будут сохранены в массиве строк. Наконец, третий вопрос: есть ли необходимость вызывать асинхронный код в методе `Main()`? Ниже мы более подробно обсудим первые два варианта, а исследование третьего отложим до главы 15.

Использование операторов верхнего уровня (нововведение в версии 9.0)

Хотя и верно то, что до выхода версии C# 9.0 все приложения .NET Core на языке C# обязаны были иметь метод `Main()`, в C# 9.0 появились операторы верхнего уровня, которые устраняют необходимость в большей части формальностей, связанных с точкой входа в приложение C#. Вы можете избавиться как от класса (`Program`), так и от метода `Main()`. Чтобы взглянуть на это в действии, приведите содержимое файла `Program.cs` к следующему виду:

```
using System;
// Отобразить пользователю простое сообщение.
Console.WriteLine("***** My First C# App *****");
Console.WriteLine("Hello World!");
Console.WriteLine();
// Ожидать нажатия клавиши <Enter>, прежде чем завершить работу.
Console.ReadLine();
```

Запустив программу, вы увидите, что получается тот же самый результат! Существует несколько правил применения операторов верхнего уровня.

- Операторы верхнего уровня можно использовать только в одном файле внутри приложения.
- В случае применения операторов верхнего уровня программа не может иметь объявленную точку входа.
- Операторы верхнего уровня нельзя помещать в пространство имен.
- Операторы верхнего уровня по-прежнему имеют доступ к строковому массиву аргументов.
- Операторы верхнего уровня возвращают код завершения приложения (как объясняется в следующем разделе) с использованием `return`.
- Функции, которые объявлялись в классе `Program`, становятся локальными функциями для операторов верхнего уровня. (Локальные функции раскрываются в главе 4.)
- Дополнительные типы можно объявлять после всех операторов верхнего уровня. Объявление любых типов до окончания операторов верхнего уровня приводит к ошибке на этапе компиляции.

"За кулисами" компилятор заполняет пробелы. Исследуя сгенерированный код IL для обновленного кода, вы заметите такое определение `TypeDef` для точки входа в приложение:

```
// TypeDef #1 (02000002)
// -------------------------------------------------------
//    TypDefName: <Program>$  (02000002)
//    Flags     : [NotPublic] [AutoLayout] [Class] [Abstract]
                  [Sealed] [AnsiClass]
      [BeforeFieldInit]  (00100180)
//    Extends   : 0100000D [TypeRef] System.Object
//    Method #1 (06000001) [ENTRYPOINT]
//    -------------------------------------------------------
//           MethodName: <Main>$ (06000001)
```

Сравните его с определением `TypeDef` для точки входа в главе 1:

```
// TypeDef #1 (02000002)
// -------------------------------------------------------
//    TypDefName: CalculatorExamples.Program  (02000002)
//    Flags     : [NotPublic] [AutoLayout] [Class] [AnsiClass]
      [BeforeFieldInit]  (00100000)
//    Extends   : 0100000C [TypeRef] System.Object
//    Method #1 (06000001) [ENTRYPOINT]
//    -------------------------------------------------------
//           MethodName: Main (06000001)
```

В примере из главы 1 обратите внимание, что значение TypDefName представлено как пространство имен (CalculatorExamples) плюс имя класса (Program), а значением MethodName является Main. В обновленном примере, использующем операторы верхнего уровня, компилятор заполняется значение <Program>$ для TypDefName и значение <Main>$ для имени метода.

Указание кода ошибки приложения (обновление в версии 9.0)

Хотя в подавляющем большинстве случаев методы Main() или операторы верхнего уровня будут иметь void в качестве возвращаемого значения, возможность возвращения int (или Task<int>) сохраняет согласованность C# с другими языками, основанными на C. По соглашению возврат значения 0 указывает на то, что программа завершилась успешно, тогда как любое другое значение (вроде -1) представляет условие ошибки (имейте в виду, что значение 0 автоматически возвращается даже в случае, если метод Main() прототипирован как возвращающий void).

При использовании операторов верхнего уровня (следовательно, в отсутствие метода Main()) в случае, если исполняемый код возвращает целое число, то оно и будет кодом возврата. Если же явно ничего не возвращается, тогда все равно обеспечивается возвращение значения 0, как при явном применении метода Main().

В ОС Windows возвращаемое приложением значение сохраняется в переменной среды по имени %ERRORLEVEL%. Если создается приложение, которое программно запускает другой исполняемый файл (тема, рассматриваемая в главе 19), тогда получить значение %ERRORLEVEL% можно с применением свойства ExitCode запущенного процесса.

Поскольку возвращаемое значение передается системе в момент завершения работы приложения, вполне очевидно, что получить и отобразить финальный код ошибки во время выполнения приложения невозможно. Однако мы покажем, как просмотреть код ошибки по завершении программы, изменив операторы верхнего уровня следующим образом:

```
// Обратите внимание, что теперь возвращается int, а не void.
// Вывести сообщение и ожидать нажатия клавиши <Enter>.
Console.WriteLine("***** My First C# App *****");
Console.WriteLine("Hello World!");
Console.WriteLine();
Console.ReadLine();

// Возвратить произвольный код ошибки.
return -1;
```

Если программа в качестве точки входа по-прежнему использует метод Main(), то вот как изменить сигнатуру метода, чтобы возвращать int вместо void:

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

Теперь давайте захватим возвращаемое значение программы с помощью пакетного файла. Используя проводник Windows, перейдите в папку, где находится файл решения (например, C:\SimpleCSharpApp), и создайте в ней новый текстовый файл (по имени SimpleCSharpApp.cmd). Поместите в файл приведенные далее инструкции (если раньше вам не приходилось создавать файлы *.cmd, то можете не беспокоиться о деталях):

```
@echo off
rem Пакетный файл для приложения SimpleCSharpApp.exe,
rem в котором захватывается возвращаемое им значение.

dotnet run
@if "%ERRORLEVEL%" == "0" goto success

:fail
  rem Приложение потерпело неудачу.
  echo This application has failed!
  echo return value = %ERRORLEVEL%
  goto end
:success
  rem Приложение завершилось успешно.
  echo This application has succeeded!
  echo return value = %ERRORLEVEL%
  goto end
:end
rem Все готово.
echo All Done.
```

Откройте окно командной подсказки (или терминал VSC) и перейдите в папку, содержащую новый файл `*.cmd`. Запустите его, набрав имя и нажав <Enter>. Вы должны получить показанный ниже вывод, учитывая, что операторы верхнего уровня или метод `Main()` возвращает `-1`. Если бы возвращалось значение `0`, то вы увидели бы в окне консоли сообщение `This application has succeeded!`.

```
***** My First C# App *****
Hello World!

This application has failed!
return value = -1
All Done.
```

Ниже приведен сценарий PowerShell, который эквивалентен предыдущему сценарию в файле `*.cmd`:

```
dotnet run
if ($LastExitCode -eq 0) {
  Write-Host "This application has succeeded!"
} else
{
  Write-Host "This application has failed!"
}
Write-Host "All Done."
```

Введите `PowerShell` в терминале VSC и запустите сценарий посредством следующей команды:

```
.\SimpleCSharpApp.ps1
```

Вот что вы увидите в терминальном окне:

```
***** My First C# App *****

Hello World!

This application has failed!
All Done.
```

В подавляющем большинстве приложений C# (если только не во всех) в качестве возвращаемого значения будет применяться void, что подразумевает неявное возвращение нулевого кода ошибки. Таким образом, все методы Main() или операторы верхнего уровня в этой книге (кроме текущего примера) будут возвращать void.

Обработка аргументов командной строки

Теперь, когда вы лучше понимаете, что собой представляет возвращаемое значение метода Main() или операторов верхнего уровня, давайте посмотрим на входной массив строковых данных. Предположим, что приложение необходимо модифицировать для обработки любых возможных параметров командной строки. Один из способов предусматривает применение цикла for языка C#. (Все итерационные конструкции языка C# более подробно рассматриваются в конце главы.)

```
// Вывести сообщение и ожидать нажатия клавиши <Enter>.
Console.WriteLine("***** My First C# App *****");
Console.WriteLine("Hello World!");
Console.WriteLine();
// Обработать любые входные аргументы.
for (int i = 0; i < args.Length; i++)
{
  Console.WriteLine("Arg: {0}", args[i]);
}
Console.ReadLine();
// Возвратить произвольный код ошибки.
return 0;
```

На заметку! В этом примере применяются операторы верхнего уровня, т.е. метод Main() не задействован. Вскоре будет показано, как обновить метод Main(), чтобы он принимал параметр args.

Снова загляните в код IL, который сгенерирован для программы, использующей операторы верхнего уровня. Обратите внимание, что метод <Main>$ принимает строковый массив по имени args, как видно ниже (для экономии пространства код приведен с сокращениями):

```
.class private abstract auto ansi sealed beforefieldinit '<Program>$'
       extends [System.Runtime]System.Object
{
  .custom instance void [System.Runtime]System.Runtime.CompilerServices.
CompilerGeneratedAttribute::.ctor()=
    ( 01 00 00 00 )
  .method private hidebysig static
         void '<Main>$'(string[] args) cil managed
  {
    .entrypoint
    ...
  } // конец метода <Program>$::<Main>$
} // конец класса <Program>$
```

Если в программе в качестве точки входа по-прежнему применяется метод Main(), тогда обеспечьте, чтобы сигнатура метода принимала строковый массив по имени args:

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

Здесь с использованием свойства `Length` класса `System.Array` производится проверка, есть ли элементы в массиве строк. Как будет показано в главе 4, все массивы C# фактически являются псевдонимом класса `System.Array` и потому разделяют общий набор членов. По мере прохода в цикле по элементам массива их значения выводятся на консоль. Предоставить аргументы в командной строке в равной степени просто:

```
C:\SimpleCSharpApp>dotnet run /arg1 -arg2
***** My First C# App *****
Hello World!
Arg: /arg1
Arg: -arg2
```

Вместо стандартного цикла `for` для реализации прохода по входному строковому массиву можно также применять ключевое слово `foreach`. Вот пример использования `foreach` (особенности конструкций циклов обсуждаются далее в главе):

```
// Обратите внимание, что в случае применения foreach
// отпадает необходимость в проверке размера массива.
foreach(string arg in args)
{
  Console.WriteLine("Arg: {0}", arg);
}
Console.ReadLine();
return 0;
```

Наконец, доступ к аргументам командной строки можно также получать с помощью статического метода `GetCommandLineArgs()` типа `System.Environment`. Данный метод возвращает массив элементов `string`. Первый элемент содержит имя самого приложения, а остальные — индивидуальные аргументы командной строки. Обратите внимание, что при таком подходе больше не обязательно определять метод `Main()` как принимающий массив `string` во входном параметре, хотя никакого вреда от этого не будет.

```
// Получить аргументы с использованием System.Environment.
string[] theArgs = Environment.GetCommandLineArgs();
foreach(string arg in theArgs)
{
  Console.WriteLine("Arg: {0}", arg);
}
Console.ReadLine();
return -1;
```

На заметку! Метод `GetCommandLineArgs()` не получает аргументы для приложения через метод `Main()` и не полагается на параметр `string[] args`.

Разумеется, именно на вас возлагается решение о том, на какие аргументы командной строки должна реагировать программа (если они вообще будут предусмотрены), и как они должны быть сформатированы (например, с префиксом - или /). В пока-

занном выше коде мы просто передаем последовательность аргументов, которые выводятся прямо в окно командной строки. Однако предположим, что создается новое игровое приложение, запрограммированное на обработку параметра вида `-godmode`. Когда пользователь запускает приложение с таким флагом, в отношении него можно было бы предпринять соответствующие действия.

Указание аргументов командной строки в Visual Studio

В реальности конечный пользователь при запуске программы имеет возможность предоставлять аргументы командной строки. Тем не менее, указывать допустимые флаги командной строки также может требоваться во время разработки в целях тестирования программы. Чтобы сделать это в Visual Studio, щелкните правой кнопкой на имени проекта в окне Solution Explorer, выберите в контекстном меню пункт Properties (Свойства), в открывшемся окне свойств перейдите на вкладку Debug (Отладка) в левой части окна, введите желаемые аргументы в текстовом поле Application arguments (Аргументы приложения) и сохраните изменения (рис. 3.1).

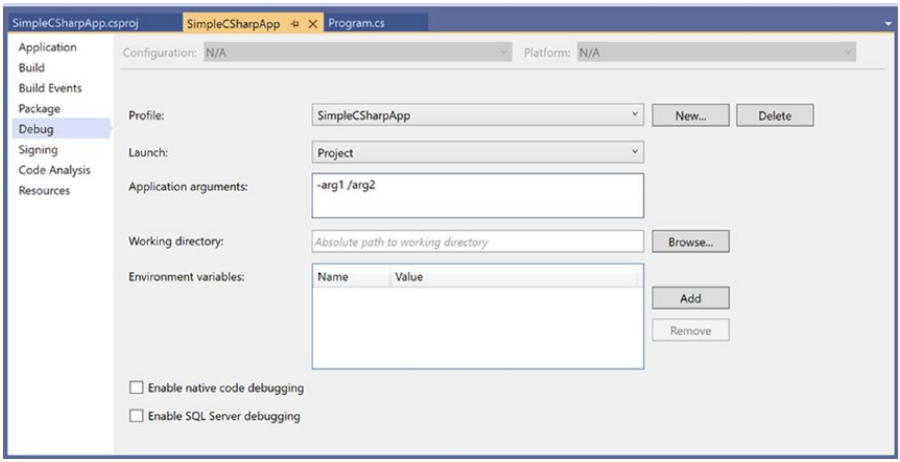

Рис. 3.1. Установка аргументов приложения в Visual Studio

Указанные аргументы командной строки будут автоматически передаваться методу `Main()` во время отладки или запуска приложения внутри IDE-среды Visual Studio.

Интересное отступление от темы: некоторые дополнительные члены класса `System.Environment`

Помимо `GetCommandLineArgs()` класс `Environment` открывает доступ к ряду других чрезвычайно полезных методов. В частности, с помощью разнообразных статических членов этот класс позволяет получать детальные сведения, касающиеся операционной системы, под управлением которой в текущий момент функционирует ваше приложение .NET 5. Для оценки полезности класса `System.Environment` измените свой код, добавив вызов локального метода по имени `ShowEnvironmentDetails()`:

```
// Локальный метод внутри операторов верхнего уровня.
ShowEnvironmentDetails();

Console.ReadLine();
return -1;
```

Реализуйте метод `ShowEnvironmentDetails()` после операторов верхнего уровня, обращаясь в нем к разным членам типа `Environment`:

```
static void ShowEnvironmentDetails()
{
  // Вывести информацию о дисковых устройствах
  // данной машины и другие интересные детали.
  foreach (string drive in Environment.GetLogicalDrives())
  {
    Console.WriteLine("Drive: {0}", drive); // Логические устройства
  }
  Console.WriteLine("OS: {0}", Environment.OSVersion);
                              // Версия операционной системы
  Console.WriteLine("Number of processors: {0}",
    Environment.ProcessorCount);          // Количество процессоров
  Console.WriteLine(".NET Core Version: {0}",
    Environment.Version);                  // Версия платформы .NET Core
}
```

Ниже показан возможный вывод, полученный в результате тестового запуска данного метода:

```
***** My First C# App *****

Hello World!

Drive: C:\
OS: Microsoft Windows NT 10.0.19042.0
Number of processors: 16
.NET Core Version: 5.0.0
```

В типе `Environment` определены и другие члены кроме тех, что задействованы в предыдущем примере. В табл. 3.1 описаны некоторые интересные дополнительные свойства; полные сведения о них можно найти в онлайновой документации.

Таблица 3.1. Избранные свойства типа `System.Environment`

Свойство	Описание
`ExitCode`	Получает или устанавливает код возврата для приложения
`Is64BitOperatingSystem`	Возвращает булевское значение, которое представляет признак наличия на текущей машине 64-разрядной операционной системы
`MachineName`	Возвращает имя текущей машины
`NewLine`	Возвращает символ новой строки для текущей среды
`SystemDirectory`	Возвращает полный путь к каталогу системы
`UserName`	Возвращает имя пользователя, запустившего данное приложение
`Version`	Возвращает объект `Version`, который представляет версию .NET Core

Использование класса System.Console

Почти во всех примерах приложений, создаваемых в начальных главах книги, будет интенсивно применяться класс System.Console. Справедливо отметить, что консольный пользовательский интерфейс может выглядеть не настолько привлекательно, как графический пользовательский интерфейс либо интерфейс веб-приложения. Однако ограничение первоначальных примеров консольными программами позволяет сосредоточиться на синтаксисе C# и ключевых аспектах платформы .NET 5, не отвлекаясь на сложности, которыми сопровождается построение настольных графических пользовательских интерфейсов или веб-сайтов.

Класс Console инкапсулирует средства манипулирования потоками ввода, вывода и ошибок для консольных приложений. В табл. 3.2 перечислены некоторые (но определенно не все) интересные его члены. Как видите, в классе Console имеется ряд членов, которые оживляют простые приложения командной строки, позволяя, например, изменять цвета фона и переднего плана и выдавать звуковые сигналы (еще и различной частоты).

Таблица 3.2. Избранные члены класса System.Console

Член	Описание
Beep()	Этот метод заставляет консоль выдать звуковой сигнал указанной частоты и длительности
BackgroundColor ForegroundColor	Эти свойства устанавливают цвета фона и переднего плана для текущего вывода. Им можно присваивать любой член перечисления ConsoleColor
BufferHeight BufferWidth	Эти свойства управляют высотой и шириной буферной области консоли
Title	Это свойство получает или устанавливает заголовок текущей консоли
WindowHeight WindowWidth WindowTop WindowLeft	Эти свойства управляют размерами консоли по отношению к установленному буферу
Clear()	Этот метод очищает установленный буфер и область отображения консоли

Выполнение базового ввода и вывода с помощью класса Console

Дополнительно к членам, описанным в табл. 3.2, в классе Console определен набор методов для захвата ввода и вывода; все они являются статическими и потому вызываются с префиксом в виде имени класса (Console). Как вы уже видели, метод WriteLine() помещает в поток вывода строку текста (включая символ возврата каретки). Метод Write() помещает в поток вывода текст без символа возврата каретки. Метод ReadLine() позволяет получить информацию из потока ввода вплоть до нажатия клавиши <Enter>. Метод Read() используется для захвата одиночного символа из потока ввода.

Чтобы реализовать базовый ввод-вывод с применением класса Console, создайте новый проект консольного приложения по имени BasicConsoleIO и добавьте его в свое решение, используя следующие команды:

```
dotnet new console -lang c# -n BasicConsoleIO -o .\BasicConsoleIO -f net5.0
dotnet sln .\Chapter3_AllProjects.sln add .\BasicConsoleIO
```

Замените код `Program.cs`, как показано ниже:

```
using System;
Console.WriteLine("***** Basic Console I/O *****");
GetUserData();
Console.ReadLine();
static void GetUserData()
{
}
```

На заметку! В Visual Studio и Visual Studio Code поддерживается несколько "фрагментов кода", которые после своей активизации вставляют код. Фрагмент кода `cw` очень полезен в начальных главах книги, т.к. он автоматически разворачивается в вызов метода `Console.WriteLine()`. Чтобы удостовериться в этом, введите `cw` где-нибудь в своем коде и нажмите клавишу <Tab>. Имейте в виду, что в Visual Studio Code клавишу <Tab> необходимо нажать один раз, а в Visual Studio — два раза.

Теперь поместите после операторов верхнего уровня реализацию метода `GetUserData()` с логикой, которая приглашает пользователя ввести некоторые сведения и затем дублирует их в стандартный поток вывода. Скажем, мы могли бы запросить у пользователя его имя и возраст (который для простоты будет трактоваться как текстовое значение, а не привычное числовое):

```
static void GetUserData()
{
  // Получить информацию об имени и возрасте.
  Console.Write("Please enter your name: "); // Предложить ввести имя
  string userName = Console.ReadLine();
  Console.Write("Please enter your age: "); // Предложить ввести возраст
  string userAge = Console.ReadLine();

  // Просто ради забавы изменить цвет переднего плана.
  ConsoleColor prevColor = Console.ForegroundColor;
  Console.ForegroundColor = ConsoleColor.Yellow;

  // Вывести полученную информацию на консоль.
  Console.WriteLine("Hello {0}! You are {1} years old.", userName, userAge);

  // Восстановить предыдущий цвет переднего плана.
  Console.ForegroundColor = prevColor;
}
```

После запуска приложения входные данные будут совершенно предсказуемо выводиться в окно консоли (с использованием указанного специального цвета).

Форматирование консольного вывода

В ходе изучения нескольких начальных глав вы могли заметить, что внутри различных строковых литералов часто встречались такие конструкции, как `{0}` и `{1}`. Платформа .NET 5 поддерживает стиль форматирования строк, который немного напоминает стиль, применяемый в операторе `printf()` языка C. Попросту говоря, когда вы определяете строковый литерал, содержащий сегменты данных, зна-

чения которых остаются неизвестными до этапа выполнения, то имеете возможность указывать заполнитель, используя синтаксис с фигурными скобками. Во время выполнения все заполнители замещаются значениями, передаваемыми методу `Console.WriteLine()`.

Первый параметр метода `WriteLine()` представляет строковый литерал, который содержит заполнители, определяемые с помощью `{0}`, `{1}`, `{2}` и т.д. Запомните, что порядковые числа заполнителей в фигурных скобках всегда начинаются с `0`. Остальные параметры `WriteLine()` — это просто значения, подлежащие вставке вместо соответствующих заполнителей.

На заметку! Если уникально нумерованных заполнителей больше, чем заполняющих аргументов, тогда во время выполнения будет сгенерировано исключение, связанное с форматом. Однако если количество заполняющих аргументов превышает число заполнителей, то лишние аргументы просто игнорируются.

Отдельный заполнитель допускается повторять внутри заданной строки. Например, если вы битломан и хотите построить строку "`9, Number 9, Number 9`", тогда могли бы написать такой код:

```
// Джон говорит...
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

Также вы должны знать о возможности помещения каждого заполнителя в любую позицию внутри строкового литерала. К тому же вовсе не обязательно, чтобы заполнители следовали в возрастающем порядке своих номеров, например:

```
// Выводит: 20, 10, 30
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```

Строки можно также форматировать с использованием интерполяции строк, которая рассматривается позже в главе.

Форматирование числовых данных

Если для числовых данных требуется более сложное форматирование, то каждый заполнитель может дополнительно содержать разнообразные символы форматирования, наиболее распространенные из которых описаны в табл. 3.3.

Таблица 3.3. Символы для форматирования числовых данных в .NET Core

Символ форматирования	Описание
`C` или `c`	Используется для форматирования денежных значений. По умолчанию значение предваряется символом локальной валюты (например, знаком доллара (`$`) для культуры US English)
`D` или `d`	Используется для форматирования десятичных чисел. В этом флаге можно также указывать минимальное количество цифр для представления значения
`E` или `e`	Используется для экспоненциального представления. Регистр этого флага указывает, в каком регистре должна представляться экспоненциальная константа — в верхнем (`E`) или в нижнем (`e`)

Окончание табл. 3.3

Символ форматирования	Описание
F или f	Используется для форматирования с фиксированной точкой. В этом флаге можно также указывать минимальное количество цифр для представления значения
G или g	Обозначает общий (general) формат. Этот флаг может использоваться для представления чисел в формате с фиксированной точкой или экспоненциальном формате
N или n	Используется для базового числового форматирования (с запятыми)
X или x	Используется для шестнадцатеричного форматирования. В случае символа X в верхнем регистре шестнадцатеричное представление будет содержать символы верхнего регистра

Символы форматирования добавляются к заполнителям в виде суффиксов после двоеточия (например, `{0:C}`, `{1:d}`, `{2:X}`). В целях иллюстрации измените метод `Main()` для вызова нового вспомогательного метода по имени `FormatNumericalData()`, реализация которого в классе `Program` форматирует фиксированное числовое значение несколькими способами.

```
// Демонстрация применения некоторых дескрипторов формата.
static void FormatNumericalData()
{
  Console.WriteLine("The value 99999 in various formats:");
  Console.WriteLine("c format: {0:c}", 99999);
  Console.WriteLine("d9 format: {0:d9}", 99999);
  Console.WriteLine("f3 format: {0:f3}", 99999);
  Console.WriteLine("n format: {0:n}", 99999);

  // Обратите внимание, что использование для символа
  // шестнадцатеричного формата верхнего или нижнего регистра
  // определяет регистр отображаемых символов.
  Console.WriteLine("E format: {0:E}", 99999);
  Console.WriteLine("e format: {0:e}", 99999);
  Console.WriteLine("X format: {0:X}", 99999);
  Console.WriteLine("x format: {0:x}", 99999);
}
```

Ниже показан вывод, получаемый в результате вызова метода `FormatNumericalData()`.

```
The value 99999 in various formats:
c format: $99,999.00
d9 format: 000099999
f3 format: 99999.000
n format: 99,999.00
E format: 9.999900E+004
e format: 9.999900e+004
X format: 1869F
x format: 1869f
```

В дальнейшем будут встречаться и другие примеры форматирования; если вас интересуют дополнительные сведения о форматировании строк, тогда обратитесь в документацию по .NET Core (`https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/formatting-types`).

Форматирование числовых данных за рамками консольных приложений

Напоследок следует отметить, что применение символов форматирования строк не ограничено консольными приложениями. Тот же самый синтаксис форматирования может быть использован при вызове статического метода `string.Format()`. Прием удобен, когда необходимо формировать выходные текстовые данные во время выполнения в приложении любого типа (например, в настольном приложении с графическим пользовательским интерфейсом, веб-приложении ASP.NET Core и т.д.).

Метод `string.Format()` возвращает новый объект `string`, который форматируется согласно предоставляемым флагам. Приведенный ниже код форматирует строку с шестнадцатеричным представлением числа:

```
// Использование string.Format() для форматирования строкового литерала.
string userMessage = string.Format("100000 in hex is {0:x}", 100000);
```

Работа с системными типами данных и соответствующими ключевыми словами C#

Подобно любому языку программирования для фундаментальных типов данных в C# определены ключевые слова, которые используются при представлении локальных переменных, переменных-членов данных в классах, возвращаемых значений и параметров методов. Тем не менее, в отличие от других языков программирования такие ключевые слова в C# являются чем-то большим, нежели просто лексемами, распознаваемыми компилятором. В действительности они представляют собой сокращенные обозначения полноценных типов из пространства имен `System`. В табл. 3.4 перечислены системные типы данных вместе с их диапазонами значений, соответствующими ключевыми словами C# и сведениями о совместимости с общеязыковой спецификацией (CLS). Все системные типы находятся в пространстве имен `System`, которое ради удобства чтения не указывается.

Таблица 3.4. Внутренние типы данных C#

Сокращенное обозначение в C#	Совместимость с CLS	Тип в System	Диапазон	Описание
`bool`	Да	`Boolean`	`true` или `false`	Признак истинности или ложности
`sbyte`	Нет	`SByte`	от –128 до 127	8-битное число со знаком
`byte`	Да	`Byte`	от 0 до 255	8-битное число без знака

Окончание табл. 3.4

Сокращенное обозначение в C#	Совместимость с CLS	Тип в System	Диапазон	Описание
short	Да	Int16	от –32 768 до 32 767	16-битное число со знаком
ushort	Нет	UInt16	от 0 до 65 535	16-битное число без знака
int	Да	Int32	от –2 147 483 648 до 2 147 483 647	32-битное число со знаком
uint	Нет	UInt32	от 0 до 4 294 967 295	32-битное число без знака
long	Да	Int64	от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	64-битное число со знаком
ulong	Нет	UInt64	от 0 до 18 446 744 073 709 551 615	64-битное число без знака
char	Да	Char	от U+0000 до U+ffff	Одиночный 16-битный символ Unicode
float	Да	Single	от -3.4×10^{38} до $+3.4\times10^{38}$	32-битное число с плавающей точкой
double	Да	Double	от $\pm5.0\times10^{-324}$ до $\pm1.7\times10^{308}$	64-битное число с плавающей точкой
decimal	Да	Decimal	(от -7.9×10^{28} до 7.9×10^{28}) / ($10^{от\ 0\ до\ 28}$)	128-битное число со знаком
string	Да	String	Ограничен объемом системной памяти	Представляет набор символов Unicode
object	Да	Object	В переменной object может храниться любой тип данных	Базовый класс для всех типов в мире .NET

На заметку! Вспомните из главы 1, что совместимый с CLS код .NET Core может быть задействован в любом другом управляемом языке программирования .NET Core. Если в программах открыт доступ к данным, не совместимым с CLS, тогда другие языки .NET Core могут быть не в состоянии их использовать.

Объявление и инициализация переменных

Для объявления локальной переменой (например, переменной внутри области видимости члена) необходимо указать тип данных, за которым следует имя переменной. Создайте новый проект консольного приложения по имени BasicDataTypes и добавьте его в свое решение с применением следующих команд:

```
dotnet new console -lang c# -n BasicDataTypes -o .\BasicDataTypes -f net5.0
dotnet sln .\Chapter3_AllProjects.sln add .\BasicDataTypes
```

Обновите код, как показано ниже:

```
using System;
using System.Numerics;

Console.WriteLine("***** Fun with Basic Data Types *****\n");
```

Теперь добавьте статическую локальную функцию `LocalVarDeclarations()` и вызовите ее в операторах верхнего уровня:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // Локальные переменные объявляются так:
  // типДанных имяПеременной;
  int myInt;
  string myString;
  Console.WriteLine();
}
```

Имейте в виду, что использование локальной переменной до присваивания ей начального значения приведет к *ошибке на этапе компиляции*. Таким образом, рекомендуется присваивать начальные значения локальным переменным непосредственно при их объявлении, что можно делать в одной строке или разносить объявление и присваивание на два отдельных оператора кода.

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // Локальные переменные объявляются и инициализируются так:
  // типДанных имяПеременной = начальноеЗначение;
  int myInt = 0;

  // Объявлять и присваивать можно также в двух отдельных строках.
  string myString;
  myString = "This is my character data";

  Console.WriteLine();
}
```

Кроме того, разрешено объявлять несколько переменных того же самого типа в одной строке кода, как в случае следующих трех переменных `bool`:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  int myInt = 0;
  string myString;
  myString = "This is my character data";

  // Объявить три переменных типа bool в одной строке.
  bool b1 = true, b2 = false, b3 = b1;
  Console.WriteLine();
}
```

Поскольку ключевое слово bool в C# — просто сокращенное обозначение структуры System.Boolean, то любой тип данных можно указывать с применением его полного имени (естественно, то же самое касается всех остальных ключевых слов C#, представляющих типы данных). Ниже приведена окончательная реализация метода LocalVarDeclarations(), в которой демонстрируются разнообразные способы объявления локальных переменных:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // Локальные переменные объявляются и инициализируются так:
  // типДанных имяПеременной = начальноеЗначение;
  int myInt = 0;
  string myString;
  myString = "This is my character data";
  // Объявить три переменных типа bool в одной строке.
  bool b1 = true, b2 = false, b3 = b1;
  // Использовать тип данных System.Boolean
  // для объявления булевской переменной.
  System.Boolean b4 = false;
  Console.WriteLine("Your data: {0}, {1}, {2}, {3}, {4}, {5}",
     myInt, myString, b1, b2, b3, b4);
  Console.WriteLine();
}
```

Литерал `default` (нововведение в версии 7.1)

Литерал default позволяет присваивать переменной стандартное значение ее типа данных. Литерал default работает для стандартных типов данных, а также для специальных классов (см. главу 5) и обобщенных типов (см. главу 10). Создайте новый метод по имени DefaultDeclarations(), поместив в него следующий код:

```
static void DefaultDeclarations()
{
  Console.WriteLine("=> Default Declarations:");
  int myInt = default;
}
```

Использование внутренних типов данных и операции `new` (обновление в версии 9.0)

Все внутренние типы данных поддерживают так называемый *стандартный конструктор* (см. главу 5). Это средство позволяет создавать переменную, используя ключевое слово new, что автоматически устанавливает переменную в ее стандартное значение:

- переменные типа bool устанавливаются в false;
- переменные числовых типов устанавливаются в 0 (или в 0.0 для типов с плавающей точкой);
- переменные типа char устанавливаются в пустой символ;
- переменные типа BigInteger устанавливаются в 0;
- переменные типа DateTime устанавливаются в 1/1/0001 12:00:00 AM;
- объектные ссылки (включая переменные типа string) устанавливаются в null.

На заметку! Тип данных `BigInteger`, упомянутый в приведенном выше списке, будет описан чуть позже.

Применение ключевого слова `new` при создании переменных базовых типов дает более громоздкий, но синтаксически корректный код C#:

```
static void NewingDataTypes()
{
  Console.WriteLine("=> Using new to create variables:");
  bool b = new bool();                  // Устанавливается в false
  int i = new int();                    // Устанавливается в 0
  double d = new double();              // Устанавливается в 0.0
  DateTime dt = new DateTime();         // Устанавливается
                                        // в 1/1/0001 12:00:00 AM
  Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
  Console.WriteLine();
}
```

В версии C# 9.0 появился сокращенный способ создания экземпляров переменных, предусматривающий применение ключевого слова `new()` без типа данных. Вот как выглядит обновленная версия предыдущего метода `NewingDataTypes()`:

```
static void NewingDataTypesWith9()
{
  Console.WriteLine("=> Using new to create variables:");
  bool b = new();             // Устанавливается в false
  int i = new();              // Устанавливается в 0
  double d = new();           // Устанавливается в 0.0
  DateTime dt = new();        // Устанавливается в 1/1/0001 12:00:00 AM
  Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
  Console.WriteLine();
}
```

Иерархия классов для типов данных

Интересно отметить, что даже элементарные типы данных в .NET Core организованы в *иерархию классов*. Если вы не знакомы с концепцией наследования, тогда найдете все необходимые сведения в главе 6. А до тех пор просто знайте, что типы, находящиеся в верхней части иерархии классов, предоставляют определенное стандартное поведение, которое передается производным типам. На рис. 3.2 показаны отношения между основными системными типами.

Обратите внимание, что каждый тип в конечном итоге оказывается производным от класса `System.Object`, в котором определен набор методов (например, `ToString()`, `Equals()`, `GetHashCode()`), общих для всех типов из библиотек базовых классов .NET Core (упомянутые методы подробно рассматриваются в главе 6).

Также важно отметить, что многие числовые типы данных являются производными от класса `System.ValueType`. Потомки `ValueType` автоматически размещаются в стеке и по этой причине имеют предсказуемое время жизни и довольно эффективны. С другой стороны, типы, в цепочке наследования которых класс `System.ValueType` отсутствует (такие как `System.Type`, `System.String`, `System.Array`, `System.Exception` и `System.Delegate`), размещаются не в стеке, а в куче с автоматической сборкой мусора. (Более подробно такое различие обсуждается в главе 4.)

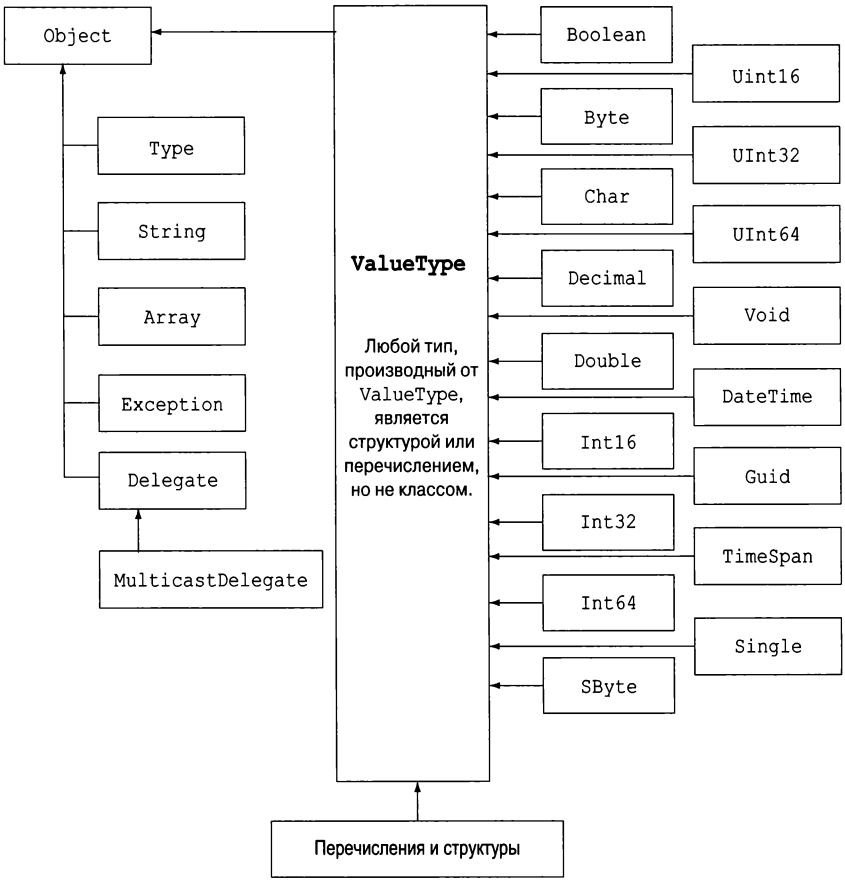

Рис. 3.2. Иерархия классов для системных типов

Не вдаваясь глубоко в детали классов `System.Object` и `System.ValueType`, важно уяснить, что поскольку любое ключевое слово C# (скажем, `int`) представляет собой просто сокращенное обозначение соответствующего системного типа (в данном случае `System.Int32`), то приведенный ниже синтаксис совершенно законен. Дело в том, что тип `System.Int32` (`int` в C#) в конечном итоге является производным от класса `System.Object` и, следовательно, может обращаться к любому из его открытых членов, как продемонстрировано в еще одной вспомогательной функции:

```
static void ObjectFunctionality()
{
  Console.WriteLine("=> System.Object Functionality:");
  // Ключевое слово int языка C# - это в действительности сокращение для
  // типа System.Int32, который наследует от System.Object следующие члены:
  Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
  Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
  Console.WriteLine("12.ToString() = {0}", 12.ToString());
```

```
  Console.WriteLine("12.GetType() = {0}", 12.GetType());
  Console.WriteLine();
}
```

Вызов метода `ObjectFunctionality()` внутри `Main()` дает такой вывод:

```
=> System.Object Functionality:
12.GetHashCode() = 12
12.Equals(23) = False
12.ToString() = 12
12.GetType() = System.Int32
```

Члены числовых типов данных

Продолжая эксперименты со встроенными типами данных C#, следует отметить, что числовые типы .NET Core поддерживают свойства `MaxValue` и `MinValue`, предоставляющие информацию о диапазоне значений, которые способен хранить конкретный тип. В дополнение к свойствам `MinValue` и `MaxValue` каждый числовой тип может определять собственные полезные члены. Например, тип `System.Double` позволяет получать значения для бесконечно малой (эпсилон) и бесконечно большой величин (которые интересны тем, кто занимается решением математических задач). В целях иллюстрации рассмотрим следующую вспомогательную функцию:

```
static void DataTypeFunctionality()
{
  Console.WriteLine("=> Data type Functionality:");
  Console.WriteLine("Max of int: {0}", int.MaxValue);
  Console.WriteLine("Min of int: {0}", int.MinValue);
  Console.WriteLine("Max of double: {0}", double.MaxValue);
  Console.WriteLine("Min of double: {0}", double.MinValue);
  Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
  Console.WriteLine("double.PositiveInfinity: {0}",
    double.PositiveInfinity);
  Console.WriteLine("double.NegativeInfinity: {0}",
    double.NegativeInfinity);
  Console.WriteLine();
}
```

В случае определения литерального целого числа (наподобие `500`) исполняющая среда по умолчанию назначит ему тип данных `int`. Аналогично литеральное число с плавающей точкой (такое как `55.333`) по умолчанию получит тип `double`. Чтобы установить тип данных в `long`, используйте суффикс `l` или `L` (`4L`). Для объявления переменной типа `float` применяйте с числовым значением суффикс `f` или `F` (`5.3F`), а для объявления десятичного числа используйте со значением с плавающей точкой суффикс `m` или `M` (`300.5M`). Это станет более важным при неявном объявлении переменных, как будет показано позже в главе.

Члены `System.Boolean`

Рассмотрим тип данных `System.Boolean`. К допустимым значениям, которые могут присваиваться типу `bool` в C#, относятся только `true` и `false`. С учетом этого должно быть понятно, что `System.Boolean` не поддерживает свойства `MinValue` и `MaxValue`, но вместо них определяет свойства `TrueString` и `FalseString` (которые выдают, соответственно, строки `"True"` и `"False"`).

Вот пример:

```
Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
Console.WriteLine("bool.TrueString: {0}", bool.TrueString);
```

Члены System.Char

Текстовые данные в C# представляются посредством ключевых слов `string` и `char`, которые являются сокращенными обозначениями для типов `System.String` и `System.Char` (оба основаны на Unicode). Как вам уже может быть известно, `string` представляет непрерывное множество символов (например, `"Hello"`), а `char` — одиночную ячейку в `string` (например, `'H'`).

Помимо возможности хранения одиночного элемента символьных данных тип `System.Char` предлагает немало другой функциональности. Используя статические методы `System.Char`, можно выяснять, является данный символ цифрой, буквой, знаком пунктуации или чем-то еще. Взгляните на следующий метод:

```
static void CharFunctionality()
{
  Console.WriteLine("=> char type Functionality:");
  char myChar = 'a';
  Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
  Console.WriteLine("char.IsLetter('a'): {0}", char.IsLetter(myChar));
  Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
    char.IsWhiteSpace("Hello There", 5));
  Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
    char.IsWhiteSpace("Hello There", 6));
  Console.WriteLine("char.IsPunctuation('?'): {0}",
    char.IsPunctuation('?'));
  Console.WriteLine();
}
```

В методе `CharFunctionality()` было показано, что для многих членов `System.Char` предусмотрены два соглашения о вызове: одиночный символ или строка с числовым индексом, указывающим позицию проверяемого символа.

Разбор значений из строковых данных

Типы данных .NET Core предоставляют возможность генерировать переменную лежащего в основе типа, имея текстовый эквивалент (например, путем выполнения разбора). Такой прием может оказаться исключительно удобным, когда вы хотите преобразовывать в числовые значения некоторые вводимые пользователем данные (вроде элемента, выбранного в раскрывающемся списке внутри графического пользовательского интерфейса). Ниже приведен пример метода `ParseFromStrings()`, содержащий логику разбора:

```
static void ParseFromStrings()
{
  Console.WriteLine("=> Data type parsing:");
  bool b = bool.Parse("True");
  Console.WriteLine("Value of b: {0}", b);  // Вывод значения b
  double d = double.Parse("99.884");
  Console.WriteLine("Value of d: {0}", d);  // Вывод значения d
  int i = int.Parse("8");
```

```
  Console.WriteLine("Value of i: {0}", i);  // Вывод значения i
  char c = Char.Parse("w");
  Console.WriteLine("Value of c: {0}", c);  // Вывод значения c
  Console.WriteLine();
}
```

Использование метода `TryParse()` для разбора значений из строковых данных

Проблема с предыдущим кодом связана с тем, что если строка не может быть аккуратно преобразована в корректный тип данных, то сгенерируется исключение. Например, следующий код потерпит неудачу во время выполнения:

```
bool b = bool.Parse("Hello");
```

Решение предусматривает помещение каждого вызова `Parse()` в блок `try-catch` (обработка исключений подробно раскрывается в главе 7), что добавит много кода, или применение метода `TryParse()`. Метод `TryParse()` принимает параметр `out` (модификатор `out` рассматривается в главе 4) и возвращает значение `bool`, которое указывает, успешно ли прошел разбор. Создайте новый метод по имени `ParseFromStringWithTryParse()` и поместите в него такой код:

```
static void ParseFromStringsWithTryParse()
{
  Console.WriteLine("=> Data type parsing with TryParse:");
  if (bool.TryParse("True", out bool b))
  {
    Console.WriteLine("Value of b: {0}", b);  // Вывод значения b
  }
  else
  {
    Console.WriteLine("Default value of b: {0}", b);
                    // Вывод стандартного значения b
  }
  string value = "Hello";
  if (double.TryParse(value, out double d))
  {
    Console.WriteLine("Value of d: {0}", d);
  }
  else
  {
    // Преобразование входного значения в double потерпело неудачу
    // и переменной было присвоено стандартное значение.
    Console.WriteLine("Failed to convert the input ({0}) to a double
and the variable was assigned the default {1}", value,d);
  }
  Console.WriteLine();
}
```

Если вы только начали осваивать программирование и не знаете, как работают операторы `if/else`, то они подробно рассматриваются позже в главе. В приведенном выше примере важно отметить, что когда строка может быть преобразована в запрошенный тип данных, метод `TryParse()` возвращает `true` и присваивает разобранное значение переменной, переданной методу. В случае невозможности разбо-

ра значения переменной присваивается стандартное значение, а метод TryParse() возвращает false.

Использование типов System.DateTime и System.TimeSpan

В пространстве имен System определено несколько полезных типов данных, для которых отсутствуют ключевые слова языка C#, в том числе структуры DateTime и TimeSpan. (При желании можете самостоятельно ознакомиться с типом System.Void, показанным на рис. 3.2.)

Тип DateTime содержит данные, представляющие специфичное значение даты (месяц, день, год) и времени, которые могут форматироваться разнообразными способами с применением членов этого типа. Структура TimeSpan позволяет легко определять и трансформировать единицы времени, используя различные ее члены.

```
static void UseDatesAndTimes()
{
  Console.WriteLine("=> Dates and Times:");

  // Этот конструктор принимает год, месяц и день.
  DateTime dt = new DateTime(2015, 10, 17);

  // Какой это день месяца?
  Console.WriteLine("The day of {0} is {1}", dt.Date, dt.DayOfWeek);

  // Сейчас месяц декабрь.
  dt = dt.AddMonths(2);
  Console.WriteLine("Daylight savings: {0}",
dt.IsDaylightSavingTime());

  // Этот конструктор принимает часы, минуты и секунды.
  TimeSpan ts = new TimeSpan(4, 30, 0);
  Console.WriteLine(ts);

  // Вычесть 15 минут из текущего значения TimeSpan и вывести результат.
  Console.WriteLine(ts.Subtract(new TimeSpan(0, 15, 0)));
}
```

Работа с пространством имен System.Numerics

В пространстве имен System.Numerics определена структура по имени BigInteger. Тип данных BigInteger может применяться для представления *огромных* числовых значений, которые не ограничены фиксированным верхним или нижним пределом.

На заметку! В пространстве имен System.Numerics также определена вторая структура по имени Complex, которая позволяет моделировать математически сложные числовые данные (например, мнимые единицы, вещественные данные, гиперболические тангенсы). Дополнительные сведения о структуре Complex можно найти в документации по .NET Core.

Несмотря на то что во многих приложениях .NET Core потребность в структуре BigInteger может никогда не возникать, если все-таки необходимо определить большое числовое значение, то в первую очередь понадобится добавить в файл показанную ниже директиву using:

```
// Здесь определен тип BigInteger:
using System.Numerics;
```

Теперь с применением операции new можно создать переменную BigInteger. Внутри конструктора можно указать числовое значение, включая данные с плавающей точкой. Однако компилятор C# неявно типизирует числа не с плавающей точкой как int, а числа с плавающей точкой — как double. Как же тогда установить для BigInteger большое значение, не переполнив стандартные типы данных, которые задействуются для неформатированных числовых значений?

Простейший подход предусматривает определение большого числового значения в виде текстового литерала, который затем может быть преобразован в переменную BigInteger посредством статического метода Parse(). При желании можно также передавать байтовый массив непосредственно конструктору класса BigInteger.

На заметку! После того как переменной BigInteger присвоено значение, модифицировать ее больше нельзя, т.к. это неизменяемые данные. Тем не менее, в классе BigInteger определено несколько членов, которые возвращают новые объекты BigInteger на основе модификаций данных (такие как статический метод Multiply(), используемый в следующем примере кода).

В любом случае после определения переменной BigInteger вы обнаружите, что в этом классе определены члены, похожие на члены в других внутренних типах данных C# (например, float либо int). Вдобавок в классе BigInteger определен ряд статических членов, которые позволяют применять к переменным BigInteger базовые математические операции (наподобие сложения и умножения). Взгляните на пример работы с классом BigInteger:

```
static void UseBigInteger()
{
  Console.WriteLine("=> Use BigInteger:");
  BigInteger biggy =
    BigInteger.Parse("99999999999999999999999999999999999999999999999");
  Console.WriteLine("Value of biggy is {0}", biggy);
                  // значение biggy
  Console.WriteLine("Is biggy an even value?: {0}", biggy.IsEven);
                  // biggy - четное?
  Console.WriteLine("Is biggy a power of two?: {0}", biggy.
IsPowerOfTwo);     // biggy - степень 2?
  BigInteger reallyBig = BigInteger.Multiply(biggy,
    BigInteger.Parse("8888888888888888888888888888888888888888888888"));
  Console.WriteLine("Value of reallyBig is {0}", reallyBig);
                  // значение reallyBig
}
```

Важно отметить, что тип данных BigInteger реагирует на внутренние математические операции C#, такие как +, - и *. Следовательно, вместо вызова метода BigInteger.Multiply() для перемножения двух больших чисел можно использовать такой код:

```
BigInteger reallyBig2 = biggy * reallyBig;
```

К настоящему моменту вы должны понимать, что ключевые слова C#, представляющие базовые типы данных, имеют соответствующие типы в библиотеках базовых классов .NET Core, каждый из которых предлагает фиксированную функциональность. Хотя абсолютно все члены этих типов данных в книге подробно не рассматриваются,

имеет смысл изучить их самостоятельно. Подробные описания разнообразных типов данных .NET Core можно найти в документации по .NET Core — скорее всего, вы будете удивлены объемом их встроенной функциональности.

Использование разделителей групп цифр (нововведение в версии 7.0)

Временами при присваивании числовой переменной крупных чисел цифр оказывается больше, чем способен отслеживать глаз. В версии C# 7.0 был введен разделитель групп цифр в виде символа подчеркивания (_) для данных `int`, `long`, `decimal`, `double` или шестнадцатеричных типов. Версия C# 7.2 позволяет шестнадцатеричным значениям (и рассматриваемым далее новым двоичным литералам) после открывающего объявления начинаться с символа подчеркивания. Ниже представлен пример применения нового разделителя групп цифр:

```
static void DigitSeparators()
{
  Console.WriteLine("=> Use Digit Separators:");
  Console.Write("Integer:");    // Целое
  Console.WriteLine(123_456);
  Console.Write("Long:");       // Длинное целое
  Console.WriteLine(123_456_789L);
  Console.Write("Float:");      // С плавающей точкой
  Console.WriteLine(123_456.1234F);
  Console.Write("Double:");     // С плавающей точкой двойной точности
  Console.WriteLine(123_456.12);
  Console.Write("Decimal:");    // Десятичное
  Console.WriteLine(123_456.12M);
  // Обновление в версии 7.2: шестнадцатеричное значение
  // может начинаться с символа _
  Console.Write("Hex:");        // Шестнадцатеричное
  Console.WriteLine(0x_00_00_FF);
}
```

Использование двоичных литералов (нововведение в версии 7.0/7.2)

В версии C# 7.0 появился новый литерал для двоичных значений, которые представляют, скажем, битовые маски. Новый разделитель групп цифр работает с двоичными литералами, а в версии C# 7.2 разрешено начинать двоичные и шестнадцатеричные числа начинать с символа подчеркивания. Теперь двоичные числа можно записывать ожидаемым образом, например:

```
0b0001_0000
```

Вот метод, в котором иллюстрируется использование новых литералов с разделителем групп цифр:

```
static void BinaryLiterals()
{
  // Обновление в версии 7.2: двоичное значение может начинаться с символа _
  Console.WriteLine("=> Use Binary Literals:");
  Console.WriteLine("Sixteen: {0}",0b_0001_0000);        // 16
  Console.WriteLine("Thirty Two: {0}",0b_0010_0000);     // 32
  Console.WriteLine("Sixty Four: {0}",0b_0100_0000);     // 64
}
```

Работа со строковыми данными

Класс `System.String` предоставляет набор членов, вполне ожидаемый от служебного класса такого рода, например, члены для возвращения длины символьных данных, поиска подстрок в текущей строке и преобразования символов между верхним и нижним регистрами. В табл. 3.5 перечислены некоторые интересные члены этого класса.

Таблица 3.5. Избранные члены класса `System.String`

Член `String`	Описание
`Length`	Свойство, которое возвращает длину текущей строки
`Compare()`	Статический метод, который позволяет сравнить две строки
`Contains()`	Метод, который позволяет определить, содержится ли в строке указанная подстрока
`Equals()`	Метод, который позволяет проверить, содержатся ли в двух строковых объектах идентичные символьные данные
`Format()`	Статический метод, позволяющий сформатировать строку с использованием других элементарных типов данных (например, числовых данных или других строк) и системы обозначений `{0}`, которая рассматривалась ранее в главе
`Insert()`	Метод, который позволяет вставить строку внутрь заданной строки
`PadLeft()` `PadRight()`	Методы, которые позволяют дополнить строку определенными символами
`Remove()` `Replace()`	Методы, которые позволяют получить копию строки с произведенными изменениями (удалением или заменой символов)
`Split()`	Метод, возвращающий массив `string`, который содержит подстроки в этом экземпляре, разделенные элементами из указанного массива `char` или `string`
`Trim()`	Метод, который удаляет все вхождения набора указанных символов с начала и конца текущей строки
`ToUpper()` `ToLower()`	Методы, которые создают копию текущей строки в верхнем или нижнем регистре

Выполнение базовых манипуляций со строками

Работа с членами `System.String` выглядит вполне ожидаемо. Просто объявите переменную `string` и задействуйте предлагаемую типом функциональность через операцию точки. Не следует забывать, что несколько членов `System.String` являются статическими и потому должны вызываться на уровне класса (а не объекта).

Создайте новый проект консольного приложения по имени `FunWithStrings` и добавьте его в свое решение. Замените существующий код следующим кодом:

```
using System;
using System.Text;
BasicStringFunctionality();
```

```
static void BasicStringFunctionality()
{
  Console.WriteLine("=> Basic String functionality:");
  string firstName = "Freddy";
                // Вывод значения firstName.
  Console.WriteLine("Value of firstName: {0}", firstName);
                // Вывод длины firstname.
  Console.WriteLine("firstName has {0} characters.", firstName.Length);
                // Вывод firstName в верхнем регистре.
  Console.WriteLine("firstName in uppercase: {0}", firstName.ToUpper());
                // Вывод firstName в нижнем регистре.
  Console.WriteLine("firstName in lowercase: {0}", firstName.ToLower());
                // Содержит ли firstName букву y?
  Console.WriteLine("firstName contains the letter y?: {0}",
    firstName.Contains("y"));
                // Вывод firstName после замены.
  Console.WriteLine("New first name: {0}", firstName.Replace("dy", ""));
  Console.WriteLine();
}
```

Здесь объяснять особо нечего: метод просто вызывает разнообразные члены, такие как `ToUpper()` и `Contains()`, на локальной переменной `string`, чтобы получить разные форматы и трансформации. Ниже приведен вывод:

```
***** Fun with Strings *****

=> Basic String functionality:
Value of firstName: Freddy
firstName has 6 characters.
firstName in uppercase: FREDDY
firstName in lowercase: freddy
firstName contains the letter y?: True
firstName after replace: Fred
```

Несмотря на то что вывод не выглядит особо неожиданным, вывод, полученный в результате вызова метода `Replace()`, может вводить в заблуждение. В действительности переменная `firstName` вообще не изменяется; взамен получается новая переменная `string` в модифицированном формате. Чуть позже мы еще вернемся к обсуждению неизменяемой природы строк.

Выполнение конкатенации строк

Переменные `string` могут соединяться вместе для построения строк большего размера с помощью операции + языка C#. Как вам должно быть известно, такой прием формально называется *конкатенацией строк*. Рассмотрим следующую вспомогательную функцию:

```
static void StringConcatenation()
{
  Console.WriteLine("=> String concatenation:");
  string s1 = "Programming the ";
  string s2 = "PsychoDrill (PTP)";
  string s3 = s1 + s2;
  Console.WriteLine(s3);
  Console.WriteLine();
}
```

Интересно отметить, что при обработке символа + компилятор C# выпускает вызов статического метода `String.Concat()`. В результате конкатенацию строк можно также выполнять, вызывая метод `String.Concat()` напрямую (хотя фактически это не дает никаких преимуществ, а лишь увеличивает объем набираемого кода):

```
static void StringConcatenation()
{
  Console.WriteLine("=> String concatenation:");
  string s1 = "Programming the ";
  string s2 = "PsychoDrill (PTP)";
  string s3 = String.Concat(s1, s2);
  Console.WriteLine(s3);
  Console.WriteLine();
}
```

Использование управляющих последовательностей

Подобно другим языкам, основанным на C, строковые литералы C# могут содержать разнообразные *управляющие последовательности*, которые позволяют уточнять то, как символьные данные должны быть представлены в потоке вывода. Каждая управляющая последовательность начинается с символа обратной косой черты, за которым следует специфический знак. В табл. 3.6 перечислены наиболее распространенные управляющие последовательности.

Таблица 3.6. Управляющие последовательности в строковых литералах

Управляющая последовательность	Описание
`\'`	Вставляет в строковый литерал символ одинарной кавычки
`\"`	Вставляет в строковый литерал символ двойной кавычки
`\\`	Вставляет в строковый литерал символ обратной косой черты. Это особенно полезно при определении путей к файлам или сетевым ресурсам
`\a`	Заставляет систему выдать звуковой сигнал, который в консольных приложениях может служить аудио-подсказкой пользователю
`\n`	Вставляет символ новой строки (на платформах Windows)
`\r`	Вставляет символ возврата каретки
`\t`	Вставляет в строковый литерал символ горизонтальной табуляции

Например, чтобы вывести строку, которая содержит символ табуляции после каждого слова, можно задействовать управляющую последовательность `\t`. Или предположим, что нужно создать один строковый литерал с символами кавычек внутри, второй — с определением пути к каталогу и третий — со вставкой трех пустых строк после вывода символьных данных. Для этого можно применять управляющие последовательности `\"`, `\\` и `\n`. Кроме того, ниже приведен еще один пример, в котором для привлечения внимания каждый строковый литерал сопровождается звуковым сигналом:

```
static void EscapeChars()
{
  Console.WriteLine("=> Escape characters:\a");
  string strWithTabs = "Model\tColor\tSpeed\tPet Name\a ";
  Console.WriteLine(strWithTabs);

  Console.WriteLine("Everyone loves \"Hello World\"\a ");
  Console.WriteLine("C:\\MyApp\\bin\\Debug\a ");

  // Добавить четыре пустых строки и снова выдать звуковой сигнал.
  Console.WriteLine("All finished.\n\n\n\a ");
  Console.WriteLine();
}
```

Выполнение интерполяции строк

Синтаксис с фигурными скобками, продемонстрированный ранее в главе ({0}, {1} и т.д.), существовал в рамках платформы .NET еще со времен версии 1.0. Начиная с выхода версии C# 6, при построении строковых литералов, содержащих заполнители для переменных, программисты на C# могут использовать альтернативный синтаксис. Формально он называется *интерполяцией строк*. Несмотря на то что выходные данные операции идентичны выходным данным, получаемым с помощью традиционного синтаксиса форматирования строк, новый подход позволяет напрямую внедрять сами переменные, а не помещать их в список с разделителями-запятыми.

Взгляните на показанный ниже дополнительный метод в нашем классе `Program` (`StringInterpolation()`), который строит переменную типа `string` с применением обоих подходов:

```
static void StringInterpolation()
{
  // Некоторые локальные переменные будут включены в крупную строку.
  int age = 4;
  string name = "Soren";

  // Использование синтаксиса с фигурными скобками.
  string greeting = string.Format("Hello {0} you are {1} years old.",
                                  name, age);

  // Использование интерполяции строк.
  string greeting2 = $"Hello {name} you are {age} years old.";
}
```

В переменной `greeting2` легко заметить, что конструируемая строка начинается с префикса `$`. Кроме того, фигурные скобки по-прежнему используются для пометки заполнителя под переменную; тем не менее, вместо применения числовой метки имеется возможность указывать непосредственно переменную. Предполагаемое преимущество заключается в том, что новый синтаксис несколько легче читать в линейной манере (слева направо) с учетом того, что не требуется "перескакивать в конец" для просмотра списка значений, подлежащих вставке во время выполнения.

С новым синтаксисом связан еще один интересный аспект: фигурные скобки, используемые в интерполяции строк, обозначают допустимую область видимости. Таким образом, с переменными можно применять операцию точки, чтобы изменять их состояние. Рассмотрим модификацию кода присваивания переменных `greeting` и `greeting2`:

```
string greeting = string.Format("Hello {0} you are {1} years old.",
                                name.ToUpper(), age);
string greeting2 = $"Hello {name.ToUpper()} you are {age} years old.";
```

Здесь посредством вызова `ToUpper()` производится преобразование значения `name` в верхний регистр. Обратите внимание, что при подходе с интерполяцией строк завершающая пара круглых скобок к вызову данного метода *не* добавляется. Учитывая это, использовать область видимости, определяемую фигурными скобками, как полноценную область видимости метода, которая содержит многочисленные строки исполняемого кода, невозможно. Взамен допускается только вызывать одиночный метод на объекте с применением операции точки, а также определять простое общее выражение наподобие `{age += 1}`.

Полезно также отметить, что в рамках нового синтаксиса внутри строкового литерала по-прежнему можно использовать управляющие последовательности. Таким образом, для вставки символа табуляции необходимо применять последовательность `\t`:

```
string greeting = string.Format("\tHello {0} you are {1} years old.",
                                name.ToUpper(), age);
string greeting2 = $"\tHello {name.ToUpper()} you are {age} years old.";
```

Определение дословных строк (обновление в версии 8.0)

Когда вы добавляете к строковому литералу префикс `@`, то создаете так называемую *дословную строку*. Используя дословные строки, вы отключаете обработку управляющих последовательностей в литералах и заставляете выводить значения `string` в том виде, как есть. Такая возможность наиболее полезна при работе со строками, представляющими пути к каталогам и сетевым ресурсам. Таким образом, вместо применения управляющей последовательности `\\` можно поступить следующим образом:

```
// Следующая строка воспроизводится дословно,
// так что отображаются все управляющие символы.
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

Также обратите внимание, что дословные строки могут использоваться для предохранения пробельных символов в строках, разнесенных по нескольким строкам вывода:

```
// При использовании дословных строк пробельные символы предохраняются.
string myLongString = @"This is a very
    very
        very
            long string";
Console.WriteLine(myLongString);
```

Применяя дословные строки, в литеральную строку можно также напрямую вставлять символы двойной кавычки, просто дублируя знак ":

```
Console.WriteLine(@"Cerebus said ""Darrr! Pret-ty sun-sets""");
```

Дословные строки также могут быть интерполированными строками за счет указания операций интерполяции (`$`) и дословности (`@`):

```
string interp = "interpolation";
string myLongString2 = $@"This is a very
  very
       long string with {interp}";
```

Нововведением в версии C# 8 является то, что порядок следования этих операций не имеет значения. Работать будет либо `$@`, либо `@$`.

Работа со строками и операциями равенства

Как будет подробно объясняться в главе 4, *ссылочный тип* — это объект, размещаемый в управляемой куче со сборкой мусора. По умолчанию при выполнении проверки на предмет равенства ссылочных типов (с помощью операций == и != языка C#) значение `true` будет возвращаться в случае, если обе ссылки указывают на один и тот же объект в памяти. Однако, несмотря на то, что тип `string` в действительности является ссылочным, операции равенства для него были переопределены так, чтобы можно было сравнивать *значения* объектов `string`, а не ссылки на объекты в памяти.

```
static void StringEquality()
{
  Console.WriteLine("=> String equality:");
  string s1 = "Hello!";
  string s2 = "Yo!";
  Console.WriteLine("s1 = {0}", s1);
  Console.WriteLine("s2 = {0}", s2);
  Console.WriteLine();

  // Проверить строки на равенство.
  Console.WriteLine("s1 == s2: {0}", s1 == s2);
  Console.WriteLine("s1 == Hello!: {0}", s1 == "Hello!");
  Console.WriteLine("s1 == HELLO!: {0}", s1 == "HELLO!");
  Console.WriteLine("s1 == hello!: {0}", s1 == "hello!");
  Console.WriteLine("s1.Equals(s2): {0}", s1.Equals(s2));
  Console.WriteLine("Yo!.Equals(s2): {0}", "Yo!".Equals(s2));
  Console.WriteLine();
}
```

Операции равенства C# выполняют в отношении объектов `string` посимвольную проверку равенства с учетом регистра и нечувствительную к культуре. Следовательно, строка `"Hello!"` не равна строке `"HELLO!"` и также отличается от строки `"hello!"`. Кроме того, памятуя о связи между `string` и `System.String`, проверку на предмет равенства можно осуществлять с использованием метода `Equals()` класса `String` и других поддерживаемых им операций равенства. Наконец, поскольку каждый строковый литерал (такой как `"Yo!"`) является допустимым экземпляром `System.String`, доступ к функциональности, ориентированной на работу со строками, можно получать для фиксированной последовательности символов.

Модификация поведения сравнения строк

Как уже упоминалось, операции равенства строк (`Compare()`, `Equals()` и ==), а также функция `IndexOf()` по умолчанию чувствительны к регистру символов и нечувствительны к культуре. Если ваша программа не заботится о регистре символов, тогда может возникнуть проблема. Один из способов ее преодоления предполагает преобразование строк в верхний или нижний регистр с последующим их сравнением:

```
if (firstString.ToUpper() == secondString.ToUpper())
{
  // Делать что-то
}
```

Здесь создается копия каждой строки со всеми символами верхнего регистра. В большинстве ситуаций это не проблема, но в случае очень крупных строк может пострадать производительность. И дело даже не производительности — написание каждый раз такого кода преобразования становится утомительным. А что, если вы забудете вызвать `ToUpper()`? Результатом будет трудная в обнаружении ошибка.

Гораздо лучший прием предусматривает применение перегруженных версий перечисленных ранее методов, которые принимают значение перечисления `StringComparison`, управляющего выполнением сравнения. Значения `StringComparison` описаны в табл. 3.7.

Таблица 3.7. Значения перечисления `StringComparison`

Операция равенства/отношения C#	Описание
`CurrentCulture`	Сравнивает строки с использованием правил сортировки, чувствительной к культуре, и текущей культуры
`CurrentCultureIgnoreCase`	Сравнивает строки с применением правил сортировки, чувствительной к культуре, и текущей культуры, игнорируя регистр символов сравниваемых строк
`InvariantCulture`	Сравнивает строки с использованием правил сортировки, чувствительной к культуре, и инвариантной культуры
`InvariantCultureIgnoreCase`	Сравнивает строки с применением правил сортировки, чувствительной к культуре, и инвариантной культуры, игнорируя регистр символов сравниваемых строк
`Ordinal`	Сравнивает строки с использованием правил ординальной (двоичной) сортировки
`OrdinalIgnoreCare`	Сравнивает строки с использованием правил ординальной (двоичной) сортировки, игнорируя регистр символов сравниваемых строк

Чтобы взглянуть на результаты применения `StringComparison`, создайте новый метод по имени `StringEqualitySpecifyingCompareRules()` со следующим кодом:

```
static void StringEqualitySpecifyingCompareRules()
{
  Console.WriteLine("=> String equality (Case Insensitive:");
  string s1 = "Hello!";
  string s2 = "HELLO!";
  Console.WriteLine("s1 = {0}", s1);
  Console.WriteLine("s2 = {0}", s2);
  Console.WriteLine();

  // Проверить результаты изменения стандартных правил сравнения.
  Console.WriteLine("Default rules: s1={0},s2={1}s1.Equals(s2): {2}",
                    s1, s2,
  s1.Equals(s2));
  Console.WriteLine("Ignore case: s1.Equals(s2,
StringComparison.OrdinalIgnoreCase): {0}",
    s1.Equals(s2, StringComparison.OrdinalIgnoreCase));
```

```
  Console.WriteLine("Ignore case, Invariant Culture: s1.Equals(s2,
     StringComparison.InvariantCultureIgnoreCase): {0}",
     s1.Equals(s2, StringComparison.InvariantCultureIgnoreCase));
  Console.WriteLine();
  Console.WriteLine("Default rules: s1={0},s2={1} s1.IndexOf(\"E\"): {2}",
    s1, s2,
    s1.IndexOf("E"));
  Console.WriteLine("Ignore case: s1.IndexOf(\"E\",
                    StringComparison.OrdinalIgnoreCase):
    {0}", s1.IndexOf("E",
    StringComparison.OrdinalIgnoreCase));
  Console.WriteLine("Ignore case, Invariant Culture: s1.IndexOf(\"E\",
    StringComparison.InvariantCultureIgnoreCase): {0}",
    s1.IndexOf("E", StringComparison.InvariantCultureIgnoreCase));
  Console.WriteLine();
}
```

В то время как приведенные здесь примеры просты и используют те же самые буквы в большинстве культур, если ваше приложение должно принимать во внимание разные наборы культур, тогда применение перечисления `StringComparison` становится обязательным.

Строки неизменяемы

Один из интересных аспектов класса `System.String` связан с тем, что после присваивания объекту `string` начального значения символьные данные *не могут быть изменены*. На первый взгляд это может показаться противоречащим действительности, ведь строкам постоянно присваиваются новые значения, а в классе `System.String` доступен набор методов, которые, похоже, только то и делают, что изменяют символьные данные тем или иным образом (скажем, преобразуя их в верхний или нижний регистр). Тем не менее, присмотревшись внимательнее к тому, что происходит "за кулисами", вы заметите, что методы типа `string` на самом деле возвращают новый объект `string` в модифицированном виде:

```
static void StringsAreImmutable()
{
  Console.WriteLine("=> Immutable Strings:\a");
  // Установить начальное значение для строки.
  string s1 = "This is my string.";
  Console.WriteLine("s1 = {0}", s1);

  // Преобразована ли строка s1 в верхний регистр?
  string upperString = s1.ToUpper();
  Console.WriteLine("upperString = {0}", upperString);

  // Нет! Строка s1 осталась в том же виде!
  Console.WriteLine("s1 = {0}", s1);
}
```

Просмотрев показанный далее вывод, можно убедиться, что в результате вызова метода `ToUpper()` исходный объект `string` (`s1`) не преобразовывался в верхний регистр. Взамен была возвращена *копия* переменной типа `string` в измененном формате.

```
=> Immutable Strings:
s1 = This is my string.
upperString = THIS IS MY STRING.
s1 = This is my string.
```

Тот же самый закон неизменяемости строк действует и в случае применения операции присваивания C#. Чтобы проиллюстрировать, реализуем следующий метод `StringsAreImmutable2()`:

```
static void StringsAreImmutable2()
{
  Console.WriteLine("=> Immutable Strings 2:\a");
  string s2 = "My other string";
  s2 = "New string value";
}
```

Скомпилируйте приложение и запустите `ildasm.exe` (см. главу 1). Ниже приведен код CIL, который будет сгенерирован для метода `StringsAreImmutable2()`:

```
.method private hidebysig static void StringsAreImmutable2() cil managed
{
  // Code size        21 (0x15)
  .maxstack  1
  .locals init (string V_0)
  IL_0000:  nop
  IL_0001:  ldstr      "My other string"
  IL_0006:  stloc.0
  IL_0007:  ldstr      "New string value" /* 70000B3B */
  IL_000c:  stloc.0
  IL_000d:  ldloc.0
  IL_0013:  nop
  IL_0014:  ret
} // end of method Program::StringsAreImmutable2
```

Хотя низкоуровневые детали языка CIL пока подробно не рассматривались, обратите внимание на многочисленные вызовы кода операции `ldstr` ("load string" — "загрузить строку"). Попросту говоря, код операции `ldstr` языка CIL загружает новый объект `string` в управляемую кучу. Предыдущий объект `string`, который содержал значение `"My other string"`, будет со временем удален сборщиком мусора.

Так что же в точности из всего этого следует? Выражаясь кратко, класс `string` может стать неэффективным и при неправильном употреблении приводить к "разбуханию" кода, особенно при выполнении конкатенации строк или при работе с большими объемами текстовых данных. Но если необходимо представлять элементарные символьные данные, такие как номер карточки социального страхования, имя и фамилия или простые фрагменты текста, используемые внутри приложения, тогда тип `string` будет идеальным вариантом.

Однако когда строится приложение, в котором текстовые данные будут часто изменяться (подобное текстовому процессору), то представление обрабатываемых текстовых данных с применением объектов `string` будет неудачным решением, т.к. оно практически наверняка (и часто косвенно) приведет к созданию излишних копий строковых данных. Тогда каким образом должен поступить программист? Ответ на этот вопрос вы найдете ниже.

Использование типа System.Text.StringBuilder

С учетом того, что тип string может оказаться неэффективным при необдуманном использовании, библиотеки базовых классов .NET Core предоставляют пространство имен System.Text. Внутри этого (относительно небольшого) пространства имен находится класс StringBuilder. Как и System.String, класс StringBuilder определяет методы, которые позволяют, например, заменять или форматировать сегменты. Для применения класса StringBuilder в файлах кода C# первым делом понадобится импортировать следующее пространство имен в файл кода (что в случае нового проекта Visual Studio уже должно быть сделано):

```
// Здесь определен класс StringBuilder:
using System.Text;
```

Уникальность класса StringBuilder в том, что при вызове его членов производится прямое изменение внутренних символьных данных объекта (делая его более эффективным) без получения копии данных в модифицированном формате. При создании экземпляра StringBuilder начальные значения объекта могут быть заданы через один из множества *конструкторов*. Если вы не знакомы с понятием конструктора, тогда пока достаточно знать только то, что конструкторы позволяют создавать объект с начальным состоянием при использовании ключевого слова new. Взгляните на следующий пример применения StringBuilder:

```
static void FunWithStringBuilder()
{
  Console.WriteLine("=> Using the StringBuilder:");
  StringBuilder sb = new StringBuilder("**** Fantastic Games ****");
  sb.Append("\n");
  sb.AppendLine("Half Life");
  sb.AppendLine("Morrowind");
  sb.AppendLine("Deus Ex" + "2");
  sb.AppendLine("System Shock");
  Console.WriteLine(sb.ToString());
  sb.Replace("2", " Invisible War");
  Console.WriteLine(sb.ToString());
  Console.WriteLine("sb has {0} chars.", sb.Length);
  Console.WriteLine();
}
```

Здесь создается объект StringBuilder с начальным значением "**** Fantastic Games ****". Как видите, можно добавлять строки в конец внутреннего буфера, а также заменять или удалять любые символы. По умолчанию StringBuilder способен хранить строку только длиной 16 символов или меньше (но при необходимости будет автоматически расширяться); однако стандартное начальное значение длины можно изменить посредством дополнительного аргумента конструктора:

```
// Создать экземпляр StringBuilder с исходным размером в 256 символов.
StringBuilder sb = new StringBuilder("**** Fantastic Games ****", 256);
```

При добавлении большего количества символов, чем в указанном лимите, объект StringBuilder скопирует свои данные в новый экземпляр и увеличит размер буфера на заданный лимит.

Сужающие и расширяющие преобразования типов данных

Теперь, когда вы понимаете, как работать с внутренними типами данных C#, давайте рассмотрим связанную тему *преобразования типов данных*. Создайте новый проект консольного приложения по имени `TypeConversions` и добавьте его в свое решение. Приведите код к следующему виду:

```
using System;
Console.WriteLine("***** Fun with type conversions *****");
// Сложить две переменные типа short и вывести результат.
short numb1 = 9, numb2 = 10;
Console.WriteLine("{0} + {1} = {2}",
  numb1, numb2, Add(numb1, numb2));
Console.ReadLine();
static int Add(int x, int y)
{
  return x + y;
}
```

Легко заметить, что метод `Add()` ожидает передачи двух параметров `int`. Тем не менее, в вызывающем коде ему на самом деле передаются две переменные типа `short`. Хотя это может выглядеть похожим на несоответствие типов данных, программа компилируется и выполняется без ошибок, возвращая ожидаемый результат 19.

Причина, по которой компилятор считает такой код синтаксически корректным, связана с тем, что потеря данных в нем невозможна. Из-за того, что максимальное значение для типа `short` (32 767) гораздо меньше максимального значения для типа `int` (2 147 483 647), компилятор неявно *расширяет* каждое значение `short` до типа `int`. Формально термин *расширение* используется для определения неявного *восходящего приведения*, которое не вызывает потерю данных.

На заметку! Разрешенные расширяющие и сужающие (обсуждаются далее) преобразования, поддерживаемые для каждого типа данных C#, описаны в разделе "Type Conversion Tables in .NET" ("Таблицы преобразования типов в .NET") документации по .NET Core.

Несмотря на то что неявное расширение типов благоприятствовало в предыдущем примере, в других ситуациях оно может стать источником ошибок на этапе компиляции. Например, пусть для переменных `numb1` и `numb2` установлены значения, которые (при их сложении) превышают максимальное значение типа `short`. Кроме того, предположим, что возвращаемое значение метода `Add()` сохраняется в новой локальной переменной `short`, а не напрямую выводится на консоль.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with type conversions *****");
  // Следующий код вызовет ошибку на этапе компиляции!
  short numb1 = 30000, numb2 = 30000;
  short answer = Add(numb1, numb2);
  Console.WriteLine("{0} + {1} = {2}",
    numb1, numb2, answer);
  Console.ReadLine();
}
```

В данном случае компилятор сообщит об ошибке:

```
Cannot implicitly convert type 'int' to 'short'. An explicit
conversion exists (are you missing a cast?)
Не удается неявно преобразовать тип int в short. Существует явное
преобразование (возможно, пропущено приведение)
```

Проблема в том, что хотя метод `Add()` способен возвратить значение `int`, равное 60 000 (которое умещается в допустимый диапазон для `System.Int32`), это значение не может быть сохранено в переменной `short`, потому что выходит за пределы диапазона допустимых значений для типа `short`. Выражаясь формально, среде CoreCLR не удалось применить *сужающую операцию*. Нетрудно догадаться, что сужающая операция является логической противоположностью расширяющей операции, поскольку предусматривает сохранение большего значения внутри переменной типа данных с меньшим диапазоном допустимых значений.

Важно отметить, что все сужающие преобразования приводят к ошибкам на этапе компиляции, даже когда есть основание полагать, что такое преобразование должно пройти успешно. Например, следующий код также вызовет ошибку при компиляции:

```
// Снова ошибка на этапе компиляции!
static void NarrowingAttempt()
{
  byte myByte = 0;
  int myInt = 200;
  myByte = myInt;
  Console.WriteLine("Value of myByte: {0}", myByte);
}
```

Здесь значение, содержащееся в переменной типа `int` (`myInt`), благополучно умещается в диапазон допустимых значений для типа `byte`; следовательно, можно было бы ожидать, что сужающая операция не должна привести к ошибке во время выполнения. Однако из-за того, что язык C# создавался с расчетом на безопасность в отношении типов, все-таки будет получена ошибка на этапе компиляции.

Если нужно проинформировать компилятор о том, что вы готовы мириться с возможной потерей данных из-за сужающей операции, тогда потребуется применить *явное приведение*, используя операцию приведения `()` языка C#. Взгляните на показанную далее модификацию класса `Program`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with type conversions *****");
    short numb1 = 30000, numb2 = 30000;
    // Явно привести int к short (и разрешить потерю данных).
    short answer = (short)Add(numb1, numb2);
    Console.WriteLine("{0} + {1} = {2}",
      numb1, numb2, answer);
    NarrowingAttempt();
    Console.ReadLine();
  }
  static int Add(int x, int y)
  {
    return x + y;
  }
```

```
  static void NarrowingAttempt()
  {
    byte myByte = 0;
    int myInt = 200;

    // Явно привести int к byte (без потери данных).
    myByte = (byte)myInt;
    Console.WriteLine("Value of myByte: {0}", myByte);
  }
}
```

Теперь компиляция кода проходит успешно, но результат сложения оказывается совершенно неправильным:

```
***** Fun with type conversions *****
30000 + 30000 = -5536
Value of myByte: 200
```

Как вы только что удостоверились, явное приведение заставляет компилятор применить сужающее преобразование, даже когда оно может вызвать потерю данных. В случае метода `NarrowingAttempt()` это не было проблемой, т.к. значение 200 умещалось в диапазон допустимых значений для типа `byte`. Тем не менее, в ситуации со сложением двух значений типа `short` внутри `Main()` конечный результат получился полностью неприемлемым (30 000 + 30 000 = –5536?).

Для построения приложений, в которых потеря данных не допускается, язык C# предлагает ключевые слова `checked` и `unchecked`, которые позволяют гарантировать, что потеря данных не останется необнаруженной.

Использование ключевого слова checked

Давайте начнем с выяснения роли ключевого слова `checked`. Предположим, что в класс `Program` добавлен новый метод, который пытается просуммировать две переменные типа `byte`, причем каждой из них было присвоено значение, не превышающее допустимый максимум (255). По идее после сложения значений этих двух переменных (с приведением результата `int` к типу `byte`) должна быть получена точная сумма.

```
static void ProcessBytes()
{
  byte b1 = 100;
  byte b2 = 250;
  byte sum = (byte)Add(b1, b2);

  // В sum должно содержаться значение 350.
  // Однако там оказывается значение 94!
  Console.WriteLine("sum = {0}", sum);
}
```

Удивительно, но при просмотре вывода приложения обнаруживается, что в переменной `sum` содержится значение 94 (а не 350, как ожидалось). Причина проста. Учитывая, что `System.Byte` может хранить только значение в диапазоне от 0 до 255 включительно, в `sum` будет помещено значение переполнения (350 – 256 = 94). По умолчанию, если не предпринимаются никакие корректирующие действия, то условия переполнения и потери значимости происходят без выдачи сообщений об ошибках.

Для обработки условий переполнения и потери значимости в приложении доступны два способа. Это можно делать вручную, полагаясь на свои знания и навыки в области программирования. Недостаток такого подхода произрастает из того факта, что мы всего лишь люди, и даже приложив максимум усилий, все равно можем попросту упустить из виду какие-то ошибки.

К счастью, язык C# предоставляет ключевое слово checked. Когда оператор (или блок операторов) помещен в контекст checked, компилятор C# выпускает дополнительные инструкции CIL, обеспечивающие проверку условий переполнения, которые могут возникать при сложении, умножении, вычитании или делении двух значений числовых типов.

Если происходит переполнение, тогда во время выполнения генерируется исключение System.OverflowException. В главе 7 будут предложены подробные сведения о структурированной обработке исключений, а также об использовании ключевых слов try и catch. Не вдаваясь пока в детали, взгляните на следующий модифицированный код:

```
static void ProcessBytes()
{
  byte b1 = 100;
  byte b2 = 250;
  // На этот раз сообщить компилятору о необходимости добавления
  // кода CIL, необходимого для генерации исключения, если возникает
  // переполнение или потеря значимости.
  try
  {
    byte sum = checked((byte)Add(b1, b2));
    Console.WriteLine("sum = {0}", sum);
  }
  catch (OverflowException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Обратите внимание, что возвращаемое значение метода Add() помещено в контекст ключевого слова checked. Поскольку значение sum выходит за пределы допустимого диапазона для типа byte, генерируется исключение времени выполнения. Сообщение об ошибке выводится посредством свойства Message:

```
Arithmetic operation resulted in an overflow.
Арифметическая операция привела к переполнению.
```

Чтобы обеспечить принудительную проверку переполнения для целого блока операторов, контекст checked можно определить так:

```
try
{
  checked
  {
    byte sum = (byte)Add(b1, b2);
    Console.WriteLine("sum = {0}", sum);
  }
}
```

```
catch (OverflowException ex)
{
  Console.WriteLine(ex.Message);
}
```

В любом случае интересующий код будет автоматически оцениваться на предмет возможных условий переполнения, и если они обнаружатся, то сгенерируется исключение, связанное с переполнением.

Настройка проверки переполнения на уровне проекта

Если создается приложение, в котором никогда не должно возникать молчаливое переполнение, то может обнаружиться, что в контекст ключевого слова `checked` приходится помещать слишком много строк кода. В качестве альтернативы компилятор C# поддерживает флаг `/checked`. Когда он указан, все присутствующие в коде арифметические операции будут оцениваться на предмет переполнения, не требуя применения ключевого слова `checked`. Если переполнение было обнаружено, тогда сгенерируется исключение времени выполнения. Чтобы установить его для всего проекта, добавьте в файл проекта следующий код:

```
<PropertyGroup>
  <CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
</PropertyGroup>
```

Настройка проверки переполнения на уровне проекта (Visual Studio)

Для активизации флага `/checked` в Visual Studio откройте окно свойств проекта. В раскрывающемся списке Configuration (Конфигурация) выберите вариант All Configurations (Все конфигурации), перейдите на вкладку Build (Сборка) и щелкните на кнопке Advanced (Дополнительно). В открывшемся диалоговом окне отметьте флажок Check for arithmetic overflow (Проверять арифметическое переполнение), как показано на рис. 3.3. Включить эту настройку может быть удобно при создании отладочной версии сборки. После устранения всех условий переполнения в кодовой базе флаг `/checked` можно отключить для последующих построений (что приведет к увеличению производительности приложения).

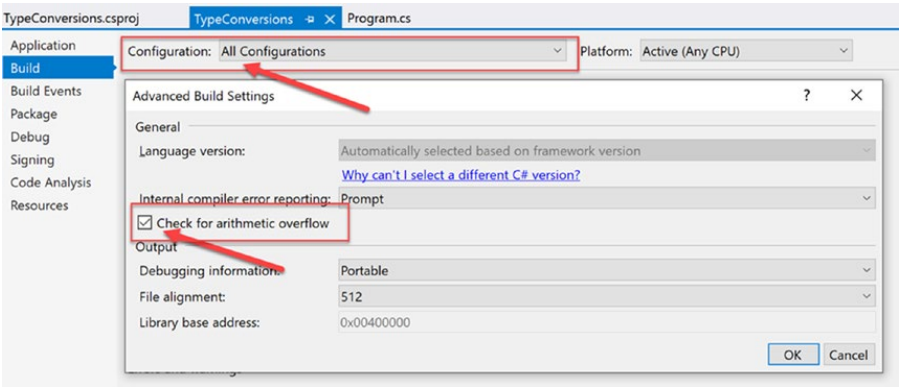

Рис. 3.3. Включение проверки переполнения в масштабах проекта

На заметку! Если вы не выберете в списке вариант All Configurations, тогда настройка будет применена только к конфигурации, выбранной в текущий момент (т.е. Debug (Отладка) или Release (Выпуск)).

Использование ключевого слова unchecked

А теперь предположим, что проверка переполнения и потери значимости включена в масштабах проекта, но есть блок кода, в котором потеря данных *приемлема*. Как с ним быть? Учитывая, что действие флага /checked распространяется на всю арифметическую логику, в языке C# имеется ключевое слово unchecked, которое предназначено для отмены генерации исключений, связанных с переполнением, в отдельных случаях. Ключевое слово unchecked используется аналогично checked, т.е. его можно применять как к единственному оператору, так и к блоку операторов:

```
// Предполагая, что флаг /checked активизирован, этот блок
// не будет генерировать исключение времени выполнения.
unchecked
{
  byte sum = (byte)(b1 + b2);
  Console.WriteLine("sum = {0} ", sum);
}
```

Подводя итоги по ключевым словам checked и unchecked в C#, следует отметить, что стандартное поведение исполняющей среды .NET Core предусматривает игнорирование арифметического переполнения и потери значимости. Когда необходимо обрабатывать избранные операторы, должно использоваться ключевое слово checked. Если нужно перехватывать ошибки переполнения по всему приложению, то придется активизировать флаг /checked. Наконец, ключевое слово unchecked может применяться при наличии блока кода, в котором переполнение приемлемо (и, следовательно, не должно приводить к генерации исключения времени выполнения).

Неявно типизированные локальные переменные

Вплоть до этого места в главе при объявлении каждой локальной переменной *явно* указывался ее тип данных:

```
static void DeclareExplicitVars()
{
  // Явно типизированные локальные переменные
  // объявляются следующим образом:
  // типДанных имяПеременной = начальноеЗначение;
  int myInt = 0;
  bool myBool = true;
  string myString = "Time, marches on...";
}
```

В то время как многие согласятся с тем, что явное указание типа данных для каждой переменной является рекомендуемой практикой, язык C# поддерживает возможность *неявной типизации* локальных переменных с использованием ключевого слова var. Ключевое слово var может применяться вместо указания конкретного типа данных (такого как int, bool или string). Когда вы поступаете подобным образом, ком-

пилятор будет автоматически выводить лежащий в основе тип данных на основе начального значения, используемого для инициализации локального элемента данных.

Чтобы прояснить роль неявной типизации, создайте новый проект консольного приложения по имени `ImplicitlyTypedLocalVars` и добавьте его в свое решение. Обновите код в `Program.cs`, как показано ниже:

```
using System;
using System.Linq;
Console.WriteLine("***** Fun with Implicit Typing *****");
```

Добавьте следующую функцию, которая демонстрирует неявные объявления:

```
static void DeclareImplicitVars()
{
  // Неявно типизированные локальные переменные
  // объявляются следующим образом:
  // var имяПеременной = начальноеЗначение;
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";
}
```

На заметку! Строго говоря, `var` не является ключевым словом языка C#. Вполне допустимо объявлять переменные, параметры и поля по имени `var`, не получая ошибок на этапе компиляции. Однако когда лексема `var` применяется в качестве типа данных, то в таком контексте она трактуется компилятором как ключевое слово.

В таком случае, основываясь на первоначально присвоенных значениях, компилятор способен вывести для переменной `myInt` тип `System.Int32`, для переменной `myBool` — тип `System.Boolean`, а для переменной `myString` — тип `System.String`. В сказанном легко убедиться за счет вывода на консоль имен типов с помощью *рефлексии*. Как будет показано в главе 17, рефлексия представляет собой действие по определению состава типа во время выполнения. Например, с помощью рефлексии можно определить тип данных неявно типизированной локальной переменной. Модифицируйте метод `DeclareImplicitVars()`:

```
static void DeclareImplicitVars()
{
  // Неявно типизированные локальные переменные.
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";
  // Вывести имена лежащих в основе типов.
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
                  // Вывод типа myInt

  Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
                  // Вывод типа myBool

  Console.WriteLine("myString is a: {0}", myString.GetType().Name);
                  // Вывод типа myString
}
```

На заметку! Имейте в виду, что такую неявную типизацию можно использовать для любых типов, включая массивы, обобщенные типы (см. главу 10) и собственные специальные типы. В дальнейшем вы увидите и другие примеры неявной типизации.

Вызов метода `DeclareImplicitVars()` в операторах верхнего уровня дает следующий вывод:

```
***** Fun with Implicit Typing *****
myInt is a: Int32
myBool is a: Boolean
myString is a: String
```

Неявное объявление чисел

Как утверждалось ранее, целые числа по умолчанию получают тип `int`, а числа с плавающей точкой — тип `double`. Создайте новый метод по имени `DeclareImplicitNumerics` и поместите в него показанный ниже код, в котором демонстрируется неявное объявление чисел:

```
static void DeclareImplicitNumerics()
{
  // Неявно типизированные числовые переменные.
  var myUInt = 0u;
  var myInt = 0;
  var myLong = 0L;
  var myDouble = 0.5;
  var myFloat = 0.5F;
  var myDecimal = 0.5M;

  // Вывод лежащего в основе типа.
  Console.WriteLine("myUInt is a: {0}", myUInt.GetType().Name);
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
  Console.WriteLine("myLong is a: {0}", myLong.GetType().Name);
  Console.WriteLine("myDouble is a: {0}", myDouble.GetType().Name);
  Console.WriteLine("myFloat is a: {0}", myFloat.GetType().Name);
  Console.WriteLine("myDecimal is a: {0}", myDecimal.GetType().Name);
}
```

Ограничения неявно типизированных переменных

С использованием ключевого слова `var` связаны разнообразные ограничения. Прежде всего, неявная типизация применима *только* к локальным переменным внутри области видимости метода или свойства. Использовать ключевое слово `var` для определения возвращаемых значений, параметров или данных полей в специальном типе не допускается. Например, показанное ниже определение класса приведет к выдаче различных сообщений об ошибках на этапе компиляции:

```
class ThisWillNeverCompile
{
  // Ошибка! Ключевое слово var не может применяться к полям!
  private var myInt = 10;
  // Ошибка! Ключевое слово var не может применяться
  // к возвращаемому значению или типу параметра!
  public var MyMethod(var x, var y){}
}
```

Кроме того, локальным переменным, которые объявлены с ключевым словом `var`, *обязано* присваиваться начальное значение в самом объявлении, причем присваивать `null` в качестве начального значения *невозможно*. Последнее ограничение должно быть рациональным, потому что на основании только `null` компилятору не удастся вывести тип, на который бы указывала переменная.

```
// Ошибка! Должно быть присвоено значение!
var myData;
// Ошибка! Значение должно присваиваться в самом объявлении!
var myInt;
myInt = 0;
// Ошибка! Нельзя присваивать null в качестве начального значения!
var myObj = null;
```

Тем не менее, присваивать `null` локальной переменной, тип которой выведен в результате начального присваивания, разрешено (при условии, что это ссылочный тип):

```
// Допустимо, если SportsCar имеет ссылочный тип!
var myCar = new SportsCar();
myCar = null;
```

Вдобавок значение неявно типизированной локальной переменной допускается присваивать другим переменным, которые типизированы как неявно, так и явно:

```
// Также нормально!
var myInt = 0;
var anotherInt = myInt;
string myString = "Wake up!";
var myData = myString;
```

Кроме того, неявно типизированную локальную переменную разрешено возвращать вызывающему коду при условии, что возвращаемый тип метода и выведенный тип переменной, определенной посредством `var`, совпадают:

```
static int GetAnInt()
{
  var retVal = 9;
  return retVal;
}
```

Неявно типизированные данные строго типизированы

Имейте в виду, что неявная типизация локальных переменных дает в результате *строго типизированные данные*. Таким образом, применение ключевого слова `var` в языке C# — *не* тот же самый прием, который используется в сценарных языках (вроде JavaScript или Perl). Кроме того, ключевое слово `var` — это не тип данных `Variant` в COM, когда переменная на протяжении своего времени жизни может хранить значения разных типов (что часто называют *динамической типизацией*).

На заметку! В C# поддерживается возможность динамической типизации с применением ключевого слова `dynamic`. Вы узнаете о таком аспекте языка в главе 18.

Взамен средство выведения типов сохраняет аспект строгой типизации языка C# и воздействует только на объявление переменных при компиляции. Затем данные трактуются, как если бы они были объявлены с выведенным типом; присваивание такой переменной значения другого типа будет приводить к ошибке на этапе компиляции.

```
static void ImplicitTypingIsStrongTyping()
{
  // Компилятору известно, что s имеет тип System.String.
  var s = "This variable can only hold string data!";
  s = "This is fine...";
  // Можно обращаться к любому члену лежащего в основе типа.
  string upper = s.ToUpper();
  // Ошибка! Присваивание числовых данных строке не допускается!
  s = 44;
}
```

Полезность неявно типизированных локальных переменных

Теперь, когда вы видели синтаксис, используемый для объявления неявно типизируемых локальных переменных, вас наверняка интересует, в каких ситуациях его следует применять. Прежде всего, использование `var` для объявления локальных переменных просто ради интереса особой пользы не принесет. Такой подход может вызвать путаницу у тех, кто будет изучать код, поскольку лишает возможности быстро определить лежащий в основе тип данных и, следовательно, затрудняет понимание общего назначения переменной. Поэтому если вы знаете, что переменная должна относиться к типу `int`, то сразу и объявляйте ее с типом `int`!

Однако, как будет показано в начале главы 13, в наборе технологий LINQ применяются *выражения запросов*, которые могут выдавать динамически создаваемые результирующие наборы, основанные на формате самого запроса. В таких случаях неявная типизация исключительно удобна, потому что вам не придется явно определять тип, который запрос может возвращать, а в ряде ситуаций это вообще невозможно. Посмотрите, сможете ли вы определить лежащий в основе тип данных `subset` в следующем примере кода LINQ?

```
static void LinqQueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
  // Запрос LINQ!
  var subset = from i in numbers where i < 10 select i;
  Console.Write("Values in subset: ");
  foreach (var i in subset)
  {
    Console.Write("{0} ", i);
  }
  Console.WriteLine();

  // К какому же типу относится subset?
  Console.WriteLine("subset is a: {0}", subset.GetType().Name);
  Console.WriteLine("subset is defined in: {0}",
                    subset.GetType().Namespace);
}
```

Вы можете предположить, что типом данных `subset` будет массив целочисленных значений. Но на самом деле он представляет собой низкоуровневый тип данных LINQ, о котором вы вряд ли что-то знаете, если только не работаете с LINQ длительное время или не откроете скомпилированный образ в утилите `ildasm.exe`. Хорошая новость в том, что при использовании LINQ вы редко (если вообще когда-либо) беспокоитесь о типе возвращаемого значения запроса; вы просто присваиваете значение неявно типизированной локальной переменной.

Фактически можно было бы даже утверждать, что *единственным случаем*, когда применение ключевого слова `var` полностью оправдано, является определение данных, возвращаемых из запроса LINQ. Запомните, если вы знаете, что нужна переменная `int`, то просто объявляйте ее с типом `int`! Злоупотребление неявной типизацией в производственном коде (через ключевое слово `var`) большинство разработчиков расценивают как плохой стиль кодирования.

Работа с итерационными конструкциями C#

Все языки программирования предлагают средства для повторения блоков кода до тех пор, пока не будет удовлетворено условие завершения. С каким бы языком вы не имели дело в прошлом, итерационные операторы C# не должны вызывать особого удивления или требовать лишь небольшого объяснения. В C# предоставляются четыре итерационные конструкции:

- цикл `for`;
- цикл `foreach/in`;
- цикл `while`;
- цикл `do/while`.

Давайте рассмотрим каждую конструкцию зацикливания по очереди, создав новый проект консольного приложения по имени `IterationsAndDecisions`.

На заметку! Материал данного раздела главы будет кратким и по существу, т.к. здесь предполагается наличие у вас опыта работы с аналогичными ключевыми словами (`if`, `for`, `switch` и т.д.) в другом языке программирования. Если нужна дополнительная информация, просмотрите темы "Iteration Statements (C# Reference)" ("Операторы итераций (справочник по C#)"), "Jump Statements (C# Reference)" ("Операторы перехода (справочник по C#)") и "Selection Statements (C# Reference)" ("Операторы выбора (справочник по C#)") в документации по C# (`https://docs.microsoft.com/ru-ru/dotnet/csharp/`).

Использование цикла `for`

Когда требуется повторять блок кода фиксированное количество раз, хороший уровень гибкости предлагает оператор `for`. В действительности вы имеете возможность указывать, сколько раз должен повторяться блок кода, а также задавать условие завершения. Не вдаваясь в излишние подробности, ниже представлен пример синтаксиса:

```
// Базовый цикл for.
static void ForLoopExample()
{
  // Обратите внимание, что переменная i видима только
  // в контексте цикла for.
  for(int i = 0; i < 4; i++)
  {
    Console.WriteLine("Number is: {0} ", i);
  }
  // Здесь переменная i больше видимой не будет.
}
```

Все трюки, которые вы научились делать в языках C, C++ и Java, по-прежнему могут использоваться при формировании операторов `for` в C#. Допускается создавать сложные условия завершения, строить бесконечные циклы и циклы в обратном направлении (посредством операции `--`), а также применять ключевые слова `goto`, `continue` и `break`.

Использование цикла `foreach`

Ключевое слово `foreach` языка C# позволяет проходить в цикле по всем элементам внутри контейнера без необходимости в проверке верхнего предела. Тем не менее, в отличие от цикла `for` цикл `foreach` будет выполнять проход по контейнеру только линейным (n+1) образом (т.е. не получится проходить по контейнеру в обратном направлении, пропускать каждый третий элемент и т.п.).

Однако если нужно просто выполнить проход по коллекции элемент за элементом, то цикл `foreach` будет великолепным выбором. Ниже приведены два примера использования цикла `foreach` — один для обхода массива строк и еще один для обхода массива целых чисел. Обратите внимание, что тип, указанный перед ключевым словом `in`, представляет тип данных контейнера.

```
// Проход по элементам массива посредством foreach.
static void ForEachLoopExample()
{
  string[] carTypes = {"Ford", "BMW", "Yugo", "Honda" };
  foreach (string c in carTypes)
  {
    Console.WriteLine(c);
  }
  int[] myInts = { 10, 20, 30, 40 };
  foreach (int i in myInts)
  {
    Console.WriteLine(i);
  }
}
```

За ключевым словом `in` может быть указан простой массив (как в приведенном примере) или, точнее говоря, любой класс, реализующий интерфейс `IEnumerable`. Как вы увидите в главе 10, библиотеки базовых классов .NET Core поставляются с несколькими коллекциями, которые содержат реализации распространенных абстрактных типов данных. Любой из них (скажем, обобщенный тип `List<T>`) может применяться внутри цикла `foreach`.

Использование неявной типизации в конструкциях `foreach`

В итерационных конструкциях `foreach` также допускается использование неявной типизации. Как и можно было ожидать, компилятор будет выводить корректный "вид типа". Вспомните пример метода LINQ, представленный ранее в главе. Даже не зная точного типа данных переменной `subset`, с применением неявной типизации все-таки можно выполнять итерацию по результирующему набору. Поместите в начало файла следующий оператор `using`:

```
using System.Linq;
static void LinqQueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // Запрос LINQ!
  var subset = from i in numbers where i < 10 select i;
  Console.Write("Values in subset: ");

  foreach (var i in subset)
  {
    Console.Write("{0} ", i);
  }
}
```

Использование циклов while и do/while

Итерационная конструкция while удобна, когда блок операторов должен выполняться до тех пор, пока не будет удовлетворено некоторое условие завершения. Внутри области видимости цикла while необходимо позаботиться о том, чтобы это условие действительно удовлетворялось, иначе получится бесконечный цикл. В следующем примере сообщение "In while loop" будет постоянно выводиться на консоль, пока пользователь не завершит цикл вводом yes в командной строке:

```
static void WhileLoopExample()
{
  string userIsDone = "";

  // Проверить копию строки в нижнем регистре.
  while(userIsDone.ToLower() != "yes")
  {
    Console.WriteLine("In while loop");
    Console.Write("Are you done? [yes] [no]: ");  // Запрос продолжения
    userIsDone = Console.ReadLine();
  }
}
```

С циклом while тесно связан оператор do/while. Подобно простому циклу while цикл do/while используется, когда какое-то действие должно выполняться неопределенное количество раз. Разница в том, что цикл do/while гарантирует, по крайней мере, однократное выполнение своего внутреннего блока кода. С другой стороны, вполне возможно, что цикл while вообще не выполнит блок кода, если условие оказывается ложным с самого начала.

```
static void DoWhileLoopExample()
{
  string userIsDone = "";
  do
  {
    Console.WriteLine("In do/while loop");
    Console.Write("Are you done? [yes] [no]: ");
    userIsDone = Console.ReadLine();
  }while(userIsDone.ToLower() != "yes");     // Обратите внимание
                                             // на точку с запятой!
}
```

Краткое обсуждение области видимости

Как и во всех языках, основанных на C (C#, Java и т.д.), *область видимости* создается с применением фигурных скобок. Вы уже видели это во многих примерах, приведенных до сих пор, включая пространства имен, классы и методы. Конструкции итерации и принятия решений также функционируют в области видимости, что иллюстрируется в примере ниже:

```
for(int i = 0; i < 4; i++)
{
  Console.WriteLine("Number is: {0} ", i);
}
```

Для таких конструкций (в предыдущем и следующем разделах) законно не использовать фигурные скобки. Другими словами, показанный далее код будет *в точности* таким же, как в примере выше:

```
for(int i = 0; i < 4; i++)
  Console.WriteLine("Number is: {0} ", i);
```

Хотя фигурные скобки разрешено не указывать, обычно поступать так — не лучшая идея. Проблема не с однострочным оператором, а с оператором, который начинается в одной строке и продолжается в нескольких строках. В отсутствие фигурных скобок можно допустить ошибки при расширении кода внутри конструкций итерации или принятия решений. Скажем, приведенные ниже два примера — *не* одинаковы:

```
for(int i = 0; i < 4; i++)
{
  Console.WriteLine("Number is: {0} ", i);
  Console.WriteLine("Number plus 1 is: {0} ", i+1)
}
for(int i = 0; i < 4; i++)
  Console.WriteLine("Number is: {0} ", i);
  Console.WriteLine("Number plus 1 is: {0} ", i+1)
```

Если вам повезет (как в этом примере), то дополнительная строка кода вызовет ошибку на этапе компиляции, поскольку переменная `i` определена только в области видимости цикла `for`. Если же не повезет, тогда вы выполните код, не помеченный как ошибка на этапе компиляции, но является логической ошибкой, которую труднее найти и устранить.

Работа с конструкциями принятия решений и операциями отношения/равенства

Теперь, когда вы умеете многократно выполнять блок операторов, давайте рассмотрим следующую связанную концепцию — управление потоком выполнения программы. Для изменения потока выполнения программы на основе разнообразных обстоятельств в C# определены две простые конструкции:

- оператор `if/else`;
- оператор `switch`.

На заметку! В версии C# 7 выражение `is` и операторы `switch` расширяются посредством приема, называемого сопоставлением с образцом. Ради полноты здесь приведены основы того, как эти расширения влияют на операторы `if/else` и `switch`. Расширения станут более понятными после чтения главы 6, где рассматриваются правила для базовых и производных классов, приведение и стандартная операция `is`.

Использование оператора `if/else`

Первым мы рассмотрим оператор `if/else`. В отличие от C и C++ оператор `if/else` в языке C# может работать только с булевскими выражениями, но не с произвольными значениями вроде `-1` и `0`.

Использование операций отношения и равенства

Обычно для получения литерального булевского значения в операторах `if/else` применяются операции, описанные в табл. 3.8.

Таблица 3.8. Операции отношения и равенства в C#

Операция отношения/ равенства	Пример использования	Описание
`==`	`if(age == 30)`	Возвращает `true`, если выражения являются одинаковыми
`!=`	`if("Foo" != myStr)`	Возвращает `true`, если выражения являются разными
`<`	`if(bonus < 2000)`	Возвращает `true`, если выражение слева (`bonus`) меньше выражения справа (`2000`)
`>`	`if(bonus > 2000)`	Возвращает `true`, если выражение слева (`bonus`) больше выражения справа (`2000`)
`<=`	`if(bonus <= 2000)`	Возвращает `true`, если выражение слева (`bonus`) меньше или равно выражению справа (`2000`)
`>=`	`if(bonus >= 2000)`	Возвращает `true`, если выражение слева (`bonus`) больше или равно выражению справа (`2000`)

И снова программисты на C и C++ должны помнить о том, что старые трюки с проверкой условия, которое включает значение, не равное нулю, в языке C# работать не будут. Пусть необходимо проверить, содержит ли текущая строка более нуля символов. У вас может возникнуть соблазн написать такой код:

```
static void IfElseExample()
{
  // Недопустимо, т.к. свойство Length возвращает int, а не bool.
  string stringData = "My textual data";
  if(stringData.Length)
  {
    // Строка длиннее 0 символов
    Console.WriteLine("string is greater than 0 characters");
  }
```

```
  else
  {
    // Строка не длиннее 0 символов
    Console.WriteLine("string is not greater than 0 characters");
  }
  Console.WriteLine();
}
```

Если вы хотите использовать свойство `String.Length` для определения истинности или ложности, тогда выражение в условии понадобится изменить так, чтобы оно давало в результате булевское значение:

```
// Допустимо, т.к. условие возвращает true или false.
if(stringData.Length > 0)
{
  Console.WriteLine("string is greater than 0 characters");
}
```

Использование операторов `if/else` и сопоставления с образцом (нововведение в версии 7.0)

В версии C# 7.0 появилась возможность применять в операторах `if/else` *сопоставление с образцом*, которое позволяет коду инспектировать объект на наличие определенных особенностей и свойств и принимать решение на основе их существования (или не существования). Не стоит беспокоиться, если вы не знакомы с объектно-ориентированным программированием; смысл предыдущего предложения станет ясен после чтения последующих глав. Пока просто имейте в виду, что вы можете проверять тип объекта с применением ключевого слова `is`, присваивать данный объект переменной в случае соответствия образцу и затем использовать эту переменную.

Метод `IfElsePatternMatching()` исследует две объектные переменные и выясняет, имеют ли они тип `string` либо `int`, после чего выводит результаты на консоль:

```
static void IfElsePatternMatching()
{
  Console.WriteLine("===If Else Pattern Matching ===/n");
  object testItem1 = 123;
  object testItem2 = "Hello";
  if (testItem1 is string myStringValue1)
  {
    Console.WriteLine($"{myStringValue1} is a string");
                    // testItem1 имеет тип string
  }
  if (testItem1 is int myValue1)
  {
    Console.WriteLine($"{myValue1} is an int"); // testItem1 имеет тип int
  }
  if (testItem2 is string myStringValue2)
  {
    Console.WriteLine($"{myStringValue2} is a string");
                    // testItem2 имеет тип string
  }
  if (testItem2 is int myValue2)
  {
    Console.WriteLine($"{myValue2} is an int"); // testItem2 имеет тип int
  }
  Console.WriteLine();
}
```

Внесение улучшений в сопоставление с образцом (нововведение в версии 9.0)

В версии C# 9.0 внесено множество улучшений в сопоставление с образцом, как показано в табл. 3.9.

Таблица 3.9. Улучшения в сопоставление с образцом

Образец	Описание
Образцы с типами	Проверяют, относится ли переменная к тому или иному типу
Образцы в круглых скобках	Усиливают или подчеркивают приоритеты сочетаний образцов
Конъюнктивные (`and`) образцы	Требуют соответствия обоим образцам
Дизъюнктивные (`or`) образцы	Требуют соответствия одному из образцов
Инвертированные (`not`) образцы	Требуют несоответствия образцу
Относительные образцы	Требуют, чтобы переменная была меньше, меньше или равна, больше, больше или равна образцу

В модифицированном методе `IfElsePatternMatchingUpdatedInCSharp9()` новые образцы демонстрируются в действии:

```
static void IfElsePatternMatchingUpdatedInCSharp9()
{
  Console.WriteLine("================ C# 9 If Else Pattern Matching
Improvements ==============/n");
  object testItem1 = 123;
  Type t = typeof(string);
  char c = 'f';
  // Образцы типов
  if (t is Type)
  {
    Console.WriteLine($"{t} is a Type");
                      // t является Type
  }
  // Относительные, конъюнктивные и дизъюнктивные образцы
  if (c is >= 'a' and <= 'z' or >= 'A' and <= 'Z')
  {
    Console.WriteLine($"{c} is a character");
                      // c является символом
  };
  // Образцы в круглых скобках
  if (c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',')
  {
    Console.WriteLine($"{c} is a character or separator");
                      // c является символом или разделителем
  };
  // Инвертированные образцы
  if (testItem1 is not string)
  {
    Console.WriteLine($"{testItem1} is not a string");
                      // c не является строкой
  }
```

```
  if (testItem1 is not null)
  {
    Console.WriteLine($"{testItem1} is not null");
                      // c не является null
  }
  Console.WriteLine();
}
```

Использование условной операции (обновление в версиях 7.2, 9.0)

Условная операция (`?:`), также называемая *тернарной условной операцией*, является сокращенным способом написания простого оператора `if/else`. Вот ее синтаксис:

```
условие ? первое_выражение : второе_выражение;
```

Условие представляет собой условную проверку (часть `if` оператора `if/else`). Если проверка проходит успешно, тогда выполняется код, следующий сразу за знаком вопроса (`?`). Если результат проверки отличается от `true`, то выполняется код, находящийся после двоеточия (часть `else` оператора `if/else`). Приведенный ранее пример кода можно было бы переписать с применением условной операции:

```
static void ExecuteIfElseUsingConditionalOperator()
{
  string stringData = "My textual data";
  Console.WriteLine(stringData.Length > 0
   ? "string is greater than 0 characters" // строка длиннее 0 символов
   : "string is not greater than 0 characters");  // строка не длиннее
                                                  // 0 символов
  Console.WriteLine();
}
```

С условной операцией связаны некоторые ограничения. Во-первых, типы конструкций `первое_выражение` и `второе_выражение` должны иметь неявные преобразования из одной в другую или, что является нововведением в версии C# 9.0, каждая обязана поддерживать неявное преобразование в целевой тип.

Во-вторых, условная операция может использоваться только в операторах присваивания. Следующий код приведет к выдаче на этапе компиляции сообщения об ошибке "Only assignment, call, increment, decrement, and new object expressions can be used as a statement" (В качестве оператора могут применяться только выражения присваивания, вызова, инкремента, декремента и создания объекта):

```
stringData.Length > 0
  ? Console.WriteLine("string is greater than 0 characters")
  : Console.WriteLine("string is not greater than 0 characters");
```

В версии C# 7.2 появилась возможность использования условной операции для возвращения ссылки на результат условия. В следующем примере задействованы две формы условной операции:

```
static void ConditionalRefExample()
{
  var smallArray = new int[] { 1, 2, 3, 4, 5 };
  var largeArray = new int[] { 10, 20, 30, 40, 50 };
```

```
    int index = 7;
    ref int refValue = ref ((index < 5)
      ? ref smallArray[index]
      : ref largeArray[index - 5]);
    refValue = 0;

    index = 2;
    ((index < 5)
      ? ref smallArray[index]
      : ref largeArray[index - 5]) = 100;

    Console.WriteLine(string.Join(" ", smallArray));
    Console.WriteLine(string.Join(" ", largeArray));
}
```

Если вы не знакомы с ключевым словом `ref`, то переживать пока не стоит, т.к. оно будет подробно раскрыто в следующей главе. В первом примере возвращается *ссылка* на местоположение массива с условием, которая присваивается переменной `refValue`. С концептуальной точки зрения считайте ссылку указателем на позицию в массиве, а не на фактическое значение, которое в ней находится. Это позволяет изменять значение в позиции массива напрямую, изменяя значение, которое присвоено переменной `refValue`. Результатом установки значения переменной `refValue` в 0 будет изменение значений второго массива: 10,20,*0*,40,50. Во втором примере значение во второй позиции первого массива изменяется на 100, давая в результате 1,2,*100*,4,5.

Использование логических операций

Для выполнения более сложных проверок оператор `if` может также включать сложные выражения и содержать операторы `else`. Синтаксис идентичен своим аналогам в языках C (C++) и Java. Для построения сложных выражений язык C# предлагает вполне ожидаемый набор логических операций, которые описан в табл. 3.10.

Таблица 3.10. Условные операции C#

Операция	Пример	Описание
`&&`	`if(age == 30 && name == "Fred")`	Операция "И". Возвращает `true`, если все выражения дают `true`
`||`	`if(age == 30 || name == "Fred")`	Операция "ИЛИ". Возвращает `true`, если хотя бы одно из выражений дает `true`
`!`	`if(!myBool)`	Операция "НЕ". Возвращает `true`, если выражение дает `false`, или `false`, если выражение дает `true`

На заметку! Операции `&&` и `||` при необходимости поддерживают сокращенный путь выполнения. Другими словами, после того, как было определено, что сложное выражение должно дать в результате `false`, оставшиеся подвыражения вычисляться не будут. Если требуется, чтобы все выражения вычислялись безотносительно к чему-либо, тогда можно использовать операции `&` и `|`.

Использование оператора switch

Еще одной простой конструкцией C# для реализации выбора является оператор switch. Как и в остальных основанных на C языках, оператор switch позволяет организовать выполнение программы на основе заранее определенного набора вариантов. Например, в следующем коде для каждого из двух возможных вариантов выводится специфичное сообщение (блок default обрабатывает недопустимый выбор):

```
// Переключение на основе числового значения.
static void SwitchExample()
{
  Console.WriteLine("1 [C#], 2 [VB]");
  Console.Write("Please pick your language preference: ");
              // Выберите предпочитаемый язык:
  string langChoice = Console.ReadLine();
  int n = int.Parse(langChoice);
  switch (n)
  {
    case 1:
      Console.WriteLine("Good choice, C# is a fine language.");
                    // Хороший выбор. C# - замечательный язык.
      break;
    case 2:
      Console.WriteLine("VB: OOP, multithreading, and more!");
                    // VB: ООП, многопоточность и многое другое!
      break;
    default:
      Console.WriteLine("Well...good luck with that!");
                    // Что ж... удачи с этим!
      break;
  }
}
```

На заметку! Язык C# требует, чтобы каждый блок case (включая default), который содержит исполняемые операторы, завершался оператором return, break или goto во избежание сквозного прохода по блокам.

Одна из замечательных особенностей оператора switch в C# связана с тем, что вдобавок к числовым значениям он позволяет оценивать данные string. На самом деле все версии C# способны оценивать типы данных char, string, bool, int, long и enum. В следующем разделе вы увидите, что в версии C# 7 появились дополнительные возможности. Вот модифицированная версия оператора switch, которая оценивает переменную типа string:

```
static void SwitchOnStringExample()
{
  Console.WriteLine("C# or VB");
  Console.Write("Please pick your language preference: ");
  string langChoice = Console.ReadLine();
  switch (langChoice.ToUpper())
  {
    case "C#":
```

```
      Console.WriteLine("Good choice, C# is a fine language.");
      break;
    case "VB":
      Console.WriteLine("VB: OOP, multithreading and more!");
      break;
    default:
      Console.WriteLine("Well...good luck with that!");
      break;
  }
}
```

Оператор `switch` также может применяться с перечислимым типом данных. Как будет показано в главе 4, ключевое слово `enum` языка C# позволяет определять специальный набор пар "имя-значение". В качестве иллюстрации рассмотрим вспомогательный метод `SwitchOnEnumExample()`, который выполняет проверку `switch` для перечисления `System.DayOfWeek`. Пример содержит ряд синтаксических конструкций, которые пока еще не рассматривались, но сосредоточьте внимание на самом использовании `switch` с типом `enum`; недостающие фрагменты будут прояснены в последующих главах.

```
static void SwitchOnEnumExample()
{
  Console.Write("Enter your favorite day of the week: ");
                // Введите любимый день недели:
  DayOfWeek favDay;
  try
  {
    favDay = (DayOfWeek) Enum.Parse(typeof(DayOfWeek), Console.ReadLine());
  }
  catch (Exception)
  {
    Console.WriteLine("Bad input!");
                    // Недопустимое входное значение!
    return;
  }
  switch (favDay)
  {
    case DayOfWeek.Sunday:
      Console.WriteLine("Football!!");
                      // Футбол!!
      break;
    case DayOfWeek.Monday:
      Console.WriteLine("Another day, another dollar.");
                      // Еще один день, еще один доллар.
      break;
    case DayOfWeek.Tuesday:
      Console.WriteLine("At least it is not Monday.");
                      // Во всяком случае, не понедельник.
      break;
    case DayOfWeek.Wednesday:
      Console.WriteLine("A fine day.");
                      // Хороший денек.
      break;
```

```
    case DayOfWeek.Thursday:
      Console.WriteLine("Almost Friday...");
                    // Почти пятница...
      break;
    case DayOfWeek.Friday:
      Console.WriteLine("Yes, Friday rules!");
                    // Да, пятница рулит!
      break;
    case DayOfWeek.Saturday:
      Console.WriteLine("Great day indeed.");
                    // Действительно великолепный день.
      break;
  }
  Console.WriteLine();
}
```

Сквозной проход от одного оператора `case` к другому оператору `case` не разрешен, но что, если множество операторов `case` должны вырабатывать тот же самый результат? К счастью, их можно комбинировать, как демонстрируется ниже:

```
case DayOfWeek.Saturday:
case DayOfWeek.Sunday:
  Console.WriteLine("It's the weekend!");
                  // Выходные!
  break;
```

Помещение любого кода между операторами `case` приведет к тому, что компилятор сообщит об ошибке. До тех пор, пока операторы `case` следуют друг за другом, как показано выше, их можно комбинировать для разделения общего кода.

В дополнение к операторам `return` и `break`, показанным в предшествующих примерах кода, оператор `switch` также поддерживает применение `goto` для выхода из условия `case` и выполнения другого оператора `case`. Несмотря на наличие поддержки, данный прием почти повсеместно считается антипаттерном и в общем случае не рекомендуется. Ниже приведен пример использования оператора `goto` в блоке `switch`:

```
static void SwitchWithGoto()
{
  var foo = 5;
  switch (foo)
  {
    case 1:
      // Делать что-то
      goto case 2;
    case 2:
      // Делать что-то другое
      break;
    case 3:
      // Еще одно действие
      goto default;
    default:
      // Стандартное действие
      break;
  }
}
```

Выполнение сопоставления с образцом в операторах `switch` (нововведение в версии 7.0, обновление в версии 9.0)

До выхода версии C# 7 сопоставляющие выражения в операторах `switch` ограничивались сравнением переменной с константными значениями, что иногда называют *образцом с константами*. В C# 7 операторы `switch` способны также задействовать *образец с типами*, при котором операторы `case` могут оценивать *тип* проверяемой переменной, и выражения `case` больше не ограничиваются константными значениями. Правило относительно того, что каждый оператор `case` должен завершаться с помощью `return` или `break`, по-прежнему остается в силе; тем не менее, операторы `goto` не поддерживают применение образца с типами.

На заметку! Если вы новичок в объектно-ориентированном программировании, тогда материал этого раздела может слегка сбивать с толку. Все прояснится в главе 6, когда мы вернемся к новым средствам сопоставления с образцом C# 7 в контексте базовых и производных классов. Пока вполне достаточно понимать, что появился мощный новый способ написания операторов `switch`.

Добавьте еще один метод по имени `ExecutePatternMatchingSwitch()` со следующим кодом:

```
static void ExecutePatternMatchingSwitch()
{
  Console.WriteLine("1 [Integer (5)], 2 [String (\"Hi\")], 3 [Decimal (2.5)]");
  Console.Write("Please choose an option: ");
  string userChoice = Console.ReadLine();
  object choice;

  // Стандартный оператор switch, в котором применяется
  // сопоставление с образцом с константами
  switch (userChoice)
  {
    case "1":
      choice = 5;
      break;
    case "2":
      choice = "Hi";
      break;
    case "3":
      choice = 2.5;
      break;
    default:
      choice = 5;
      break;
  }

  // Новый оператор switch, в котором применяется
  // сопоставление с образцом с типами
  switch (choice)
  {
    case int i:
      Console.WriteLine("Your choice is an integer.");
                        // Выбрано целое число
```

```
      break;
    case string s:
      Console.WriteLine("Your choice is a string.");
                    // Выбрана строка
      break;
    case decimal d:
      Console.WriteLine("Your choice is a decimal.");
                    // Выбрано десятичное число
      break;
    default:
      Console.WriteLine("Your choice is something else");
                    // Выбрано что-то другое
      break;
  }
  Console.WriteLine();
}
```

В первом операторе `switch` используется стандартный образец с константами; он включен только ради полноты этого (тривиального) примера. Во втором операторе `switch` переменная типизируется как `object` и на основе пользовательского ввода может быть разобрана в тип данных `int`, `string` или `decimal`. В зависимости от *типа* переменной совпадения дают разные операторы `case`. Вдобавок к проверке типа данных в каждом операторе `case` выполняется присваивание переменной (кроме случая `default`). Модифицируйте код, чтобы задействовать значения таких переменных:

```
// Новый оператор switch, в котором применяется
// сопоставление с образцом с типами
switch (choice)
{
  case int i:
    Console.WriteLine("Your choice is an integer {0}.", i);
    break;
  case string s:
    Console.WriteLine("Your choice is a string {0}.", s);
    break;
  case decimal d:
    Console.WriteLine("Your choice is a decimal {0}.", d);
    break;
  default:
    Console.WriteLine("Your choice is something else.");
    break;
}
```

Кроме оценки типа сопоставляющего выражения к операторам `case` могут быть добавлены конструкции `when` для оценки условий на переменной. В представленном ниже примере в дополнение к проверке типа производится проверка на совпадение преобразованного типа:

```
static void ExecutePatternMatchingSwitchWithWhen()
{
  Console.WriteLine("1 [C#], 2 [VB]");
  Console.Write("Please pick your language preference: ");
```

```
  object langChoice = Console.ReadLine();
  var choice = int.TryParse(langChoice.ToString(),
                            out int c) ? c : langChoice;
  switch (choice)
  {
    case int i when i == 2:
    case string s when s.Equals("VB",
      StringComparison.OrdinalIgnoreCase):
      Console.WriteLine("VB: OOP, multithreading, and more!");
                    // VB: ООП, многопоточность и многое другое!
      break;
    case int i when i == 1:
    case string s when s.Equals("C#",
                                StringComparison.OrdinalIgnoreCase):
      Console.WriteLine("Good choice, C# is a fine language.");
                    // Хороший выбор. C# - замечательный язык.
      break;
    default:
      Console.WriteLine("Well...good luck with that!");
                    // Хорошо, удачи с этим!
      break;
  }
  Console.WriteLine();
}
```

Здесь к оператору `switch` добавляется новое измерение, поскольку порядок следования операторов `case` теперь важен. При использовании образца с константами каждый оператор `case` обязан быть уникальным. В случае применения образца с типами это больше не так. Например, следующий код будет давать совпадение для каждого целого числа в первом операторе `case`, а второй и третий оператор `case` никогда не выполнятся (на самом деле такой код даже не скомпилируется):

```
switch (choice)
{
  case int i:
    // Делать что-то
    break;
  case int i when i == 0:
    // Делать что-то
    break;
  case int i when i == -1:
    // Делать что-то
    break;
}
```

В первоначальном выпуске C# 7 возникало небольшое затруднение при сопоставлении с образцом, когда в нем использовались обобщенные типы. В версии C# 7.1 проблема была устранена. Обобщенные типы рассматриваются в главе 10.

На заметку! Все продемонстрированные ранее улучшения сопоставления с образцом в C# 9.0 также можно применять в операторах `switch`.

Использование выражений `switch` (нововведение в версии 8.0)

В версии C# 8 появились выражения `switch`, позволяющие присваивать значение переменной в лаконичном операторе. Рассмотрим версию C# 7 метода `FromRainbowClassic()`, который принимает имя цвета и возвращает для него шестнадцатеричное значение:

```
static string FromRainbowClassic(string colorBand)
{
  switch (colorBand)
  {
    case "Red":
      return "#FF0000";
    case "Orange":
      return "#FF7F00";
    case "Yellow":
      return "#FFFF00";
    case "Green":
      return "#00FF00";
    case "Blue":
      return "#0000FF";
    case "Indigo":
      return "#4B0082";
    case "Violet":
      return "#9400D3";
    default:
      return "#FFFFFF";
  };
}
```

С помощью новых выражений `switch` в C# 8 код предыдущего метода можно переписать следующим образом, сделав его гораздо более лаконичным:

```
static string FromRainbow(string colorBand)
{
  return colorBand switch
  {
    "Red" => "#FF0000",
    "Orange" => "#FF7F00",
    "Yellow" => "#FFFF00",
    "Green" => "#00FF00",
    "Blue" => "#0000FF",
    "Indigo" => "#4B0082",
    "Violet" => "#9400D3",
    _ => "#FFFFFF",
  };
}
```

В приведенном примере присутствует много непонятного, начиная с лямбда-операции (=>) и заканчивая отбрасыванием (_). Все это будет раскрыто позже в книге и данный пример окончательно прояснится.

Перед тем, как завершить обсуждение темы выражений `switch`, давайте рассмотрим еще один пример, в котором вовлечены кортежи. Кортежи подробно раскрываются в главе 4, а пока считайте кортеж простой конструкцией, которая содержит более одного значения и определяется посредством круглых скобок, подобно следующему кортежу, содержащему значения `string` и `int`:

```
(string, int)
```

В показанном ниже примере два значения, передаваемые методу `RockPaperScissors()`, преобразуются в кортеж, после чего выражение `switch` вычисляет два значения в единственном выражении. Такой прием позволяет сравнивать в операторе `switch` более одного выражения:

```
// Выражения switch с кортежами.
static string RockPaperScissors(string first, string second)
{
  return (first, second) switch
  {
    ("rock", "paper") => "Paper wins.",
    ("rock", "scissors") => "Rock wins.",
    ("paper", "rock") => "Paper wins.",
    ("paper", "scissors") => "Scissors wins.",
    ("scissors", "rock") => "Rock wins.",
    ("scissors", "paper") => "Scissors wins.",
    (_, _) => "Tie.",
  };
}
```

Чтобы вызвать метод `RockPaperScissors()`, добавьте в метод `Main()` следующие строки кода:

```
Console.WriteLine(RockPaperScissors("paper","rock"));
Console.WriteLine(RockPaperScissors("scissors","rock"));
```

Мы еще вернемся к этому примеру в главе 4, где будут представлены кортежи.

Резюме

Цель настоящей главы заключалась в демонстрации многочисленных ключевых аспектов языка программирования C#. Мы исследовали привычные конструкции, которые могут быть задействованы при построении любого приложения. После ознакомления с ролью объекта приложения вы узнали о том, что каждая исполняемая программа на C# должна иметь тип, определяющий метод `Main()`, либо явно, либо с использованием операторов верхнего уровня. Данный метод служит точкой входа в программу.

Затем были подробно описаны встроенные типы данных C# и разъяснено, что применяемые для их представления ключевые слова (например, `int`) на самом деле являются сокращенными обозначениями полноценных типов из пространства имен `System` (`System.Int32` в данном случае). С учетом этого каждый тип данных C# имеет набор встроенных членов. Кроме того, обсуждалась роль расширения и сужения, а также ключевых слов `checked` и `unchecked`.

В завершение главы рассматривалась роль неявной типизации с использованием ключевого слова `var`. Как было отмечено, неявная типизация наиболее полезна при работе с моделью программирования LINQ. Наконец, мы бегло взглянули на различные конструкции C#, предназначенные для организации циклов и принятия решений.

Теперь, когда вы понимаете некоторые базовые механизмы, в главе 4 завершится исследование основных средств языка. После этого вы будете хорошо подготовлены к изучению объектно-ориентированных возможностей C#, которое начнется в главе 5.

ГЛАВА 4

Главные конструкции программирования на C#: часть 2

В настоящей главе завершается обзор основных аспектов языка программирования C#, который был начат в главе 3. Первым делом мы рассмотрим детали манипулирования массивами с использованием синтаксиса C# и продемонстрируем функциональность, содержащуюся внутри связанного класса `System.Array`.

Далее мы выясним различные подробности, касающиеся построения методов, за счет исследования ключевых слов `out`, `ref` и `params`. В ходе дела мы объясним роль необязательных и именованных параметров. Обсуждение темы методов завершится *перегрузкой методов*.

Затем будет показано, как создавать типы перечислений и структур, включая детальное исследование отличий между *типами значений* и *ссылочными типами*. В конце главы объясняется роль типов данных, допускающих `null`, и связанных с ними операций.

После освоения материала главы вы можете смело переходить к изучению объектно-ориентированных возможностей языка C#, рассмотрение которых начнется в главе 5.

Понятие массивов C#

Как вам уже наверняка известно, *массив* — это набор элементов данных, для доступа к которым применяется числовой индекс. Выражаясь более конкретно, массив представляет собой набор расположенных рядом элементов данных одного и того же типа (массив элементов `int`, массив элементов `string`, массив элементов `SportsCar` и т.д.). Объявлять, заполнять и получать доступ к массиву в языке C# довольно просто. В целях иллюстрации создайте новый проект консольного приложения по имени `FunWithArrays`, содержащий вспомогательный метод `SimpleArrays()`:

```
Console.WriteLine("***** Fun with Arrays *****");
SimpleArrays();
Console.ReadLine();

static void SimpleArrays()
{
  Console.WriteLine("=> Simple Array Creation.");
```

```
  // Создать и заполнить массив из 3 целых чисел.
  int[] myInts = new int[3];

  // Создать строковый массив из 100 элементов с индексами 0 - 99.
  string[] booksOnDotNet = new string[100];
  Console.WriteLine();
}
```

Внимательно взгляните на комментарии в коде. При объявлении массива C# с использованием подобного синтаксиса число, указанное в объявлении, обозначает общее количество элементов, а не верхнюю границу. Кроме того, нижняя граница в массиве всегда начинается с 0. Таким образом, в результате записи `int[] myInts = new int[3]` получается массив, который содержит три элемента, проиндексированные по позициям 0, 1, 2.

После определения переменной массива можно переходить к заполнению элементов от индекса к индексу, как показано ниже в модифицированном методе `SimpleArrays()`:

```
static void SimpleArrays()
{
  Console.WriteLine("=> Simple Array Creation.");

  // Создать и заполнить массив из трех целочисленных значений.
  int[] myInts = new int[3];
  myInts[0] = 100;
  myInts[1] = 200;
  myInts[2] = 300;

  // Вывести все значения.
  foreach(int i in myInts)
  {
    Console.WriteLine(i);
  }
  Console.WriteLine();
}
```

На заметку! Имейте в виду, что если массив объявлен, но его элементы явно не заполнены по каждому индексу, то они получат стандартное значение для соответствующего типа данных (например, элементы массива `bool` будут установлены в `false`, а элементы массива `int` — в 0).

Синтаксис инициализации массивов C#

В дополнение к заполнению массива элемент за элементом есть также возможность заполнять его с применением *синтаксиса инициализации массивов*. Для этого понадобится указать значения всех элементов массива в фигурных скобках (`{}`). Такой синтаксис удобен при создании массива известного размера, когда нужно быстро задать его начальные значения. Например, вот как выглядят альтернативные версии объявления массива:

```
static void ArrayInitialization()
{
  Console.WriteLine("=> Array Initialization.");

  // Синтаксис инициализации массивов с использованием ключевого слова new.
```

```
  string[] stringArray = new string[]
    { "one", "two", "three" };
  Console.WriteLine("stringArray has {0} elements", stringArray.Length);

  // Синтаксис инициализации массивов без использования
  // ключевого слова new.
  bool[] boolArray = { false, false, true };
  Console.WriteLine("boolArray has {0} elements", boolArray.Length);

  // Инициализация массива с применением ключевого слова new
  // и указанием размера.
  int[] intArray = new int[4] { 20, 22, 23, 0 };
  Console.WriteLine("intArray has {0} elements", intArray.Length);
  Console.WriteLine();
}
```

Обратите внимание, что в случае использования синтаксиса с фигурными скобками нет необходимости указывать размер массива (как видно на примере создания переменной `stringArray`), поскольку размер автоматически вычисляется на основе количества элементов внутри фигурных скобок. Кроме того, применять ключевое слово `new` не обязательно (как при создании массива `boolArray`).

В случае объявления `intArray` снова вспомните, что указанное числовое значение представляет количество элементов в массиве, а не верхнюю границу. Если объявленный размер и количество инициализаторов не совпадают (инициализаторов слишком много или не хватает), тогда на этапе компиляции возникнет ошибка. Пример представлен ниже:

```
// Несоответствие размера и количества элементов!
int[] intArray = new int[2] { 20, 22, 23, 0 };
```

Понятие неявно типизированных локальных массивов

В главе 3 рассматривалась тема неявно типизированных локальных переменных. Как вы помните, ключевое слово `var` позволяет определять переменную, тип которой выводится компилятором. Аналогичным образом ключевое слово `var` можно использовать для определения *неявно типизированных локальных массивов*. Такой подход позволяет выделять память под новую переменную массива, не указывая тип элементов внутри массива (обратите внимание, что применение этого подхода предусматривает обязательное использование ключевого слова `new`):

```
static void DeclareImplicitArrays()
{
  Console.WriteLine("=> Implicit Array Initialization.");

  // Переменная a на самом деле имеет тип int[].
  var a = new[] { 1, 10, 100, 1000 };
  Console.WriteLine("a is a: {0}", a.ToString());

  // Переменная b на самом деле имеет тип double[].
  var b = new[] { 1, 1.5, 2, 2.5 };
  Console.WriteLine("b is a: {0}", b.ToString());

  // Переменная c на самом деле имеет тип string[].
  var c = new[] { "hello", null, "world" };
  Console.WriteLine("c is a: {0}", c.ToString());
  Console.WriteLine();
}
```

Разумеется, как и при создании массива с применением явного синтаксиса C#, элементы в списке инициализации массива должны принадлежать одному и тому же типу (например, должны быть все int, все string или все SportsCar). В отличие от возможных ожиданий, неявно типизированный локальный массив не получает по умолчанию тип System.Object, так что следующий код приведет к ошибке на этапе компиляции:

```
// Ошибка! Смешанные типы!
var d = new[] { 1, "one", 2, "two", false };
```

Определение массива объектов

В большинстве случаев массив определяется путем указания явного типа элементов, которые могут в нем содержаться. Хотя это выглядит довольно прямолинейным, существует одна важная особенность. Как будет показано в главе 6, изначальным базовым классом для каждого типа (включая фундаментальные типы данных) в системе типов .NET Core является System.Object. С учетом такого факта, если определить массив типа данных System.Object, то его элементы могут представлять все что угодно. Взгляните на следующий метод ArrayOfObjects():

```
static void ArrayOfObjects()
{
  Console.WriteLine("=> Array of Objects.");

  // Массив объектов может содержать все что угодно.
  object[] myObjects = new object[4];
  myObjects[0] = 10;
  myObjects[1] = false;
  myObjects[2] = new DateTime(1969, 3, 24);
  myObjects[3] = "Form & Void";
  foreach (object obj in myObjects)
  {
    // Вывести тип и значение каждого элемента в массиве.
    Console.WriteLine("Type: {0}, Value: {1}", obj.GetType(), obj);
  }
  Console.WriteLine();
}
```

Здесь во время прохода по содержимому массива myObjects для каждого элемента выводится лежащий в основе тип, получаемый с помощью метода GetType() класса System.Object, и его значение.

Не вдаваясь пока в детали работы метода System.Object.GetType(), просто отметим, что он может использоваться для получения полностью заданного имени элемента (службы извлечения информации о типах и рефлексии исследуются в главе 17). Приведенный далее вывод является результатом вызова метода ArrayOfObjects():

```
=> Array of Objects.
Type: System.Int32, Value: 10
Type: System.Boolean, Value: False
Type: System.DateTime, Value: 3/24/1969 12:00:00 AM
Type: System.String, Value: Form & Void
```

Работа с многомерными массивами

В дополнение к одномерным массивам, которые вы видели до сих пор, в языке C# поддерживаются два вида многомерных массивов. Первый вид называется *прямоугольным массивом*, который имеет несколько измерений, а содержащиеся в нем строки обладают одной и той же длиной. Прямоугольный многомерный массив объявляется и заполняется следующим образом:

```
static void RectMultidimensionalArray()
{
  Console.WriteLine("=> Rectangular multidimensional array.");
  // Прямоугольный многомерный массив.
  int[,] myMatrix;
  myMatrix = new int[3,4];

  // Заполнить массив (3 * 4).
  for(int i = 0; i < 3; i++)
  {
    for(int j = 0; j < 4; j++)
    {
      myMatrix[i, j] = i * j;
    }
  }

  // Вывести содержимое массива (3 * 4).
  for(int i = 0; i < 3; i++)
  {
    for(int j = 0; j < 4; j++)
    {
      Console.Write(myMatrix[i, j] + "\t");
    }
    Console.WriteLine();
  }
  Console.WriteLine();
}
```

Второй вид многомерных массивов носит название *зубчатого* (или *ступенчатого*) *массива*. Такой массив содержит какое-то число внутренних массивов, каждый из которых может иметь отличающийся верхний предел. Вот пример:

```
static void JaggedMultidimensionalArray()
{
  Console.WriteLine("=> Jagged multidimensional array.");
  // Зубчатый многомерный массив (т.е. массив массивов).
  // Здесь мы имеем массив из 5 разных массивов.
  int[][] myJagArray = new int[5][];

  // Создать зубчатый массив.
  for (int i = 0; i < myJagArray.Length; i++)
  {
    myJagArray[i] = new int[i + 7];
  }

  // Вывести все строки (помните, что каждый элемент имеет
  // стандартное значение 0).
  for(int i = 0; i < 5; i++)
  {
```

```
    for(int j = 0; j < myJagArray[i].Length; j++)
    {
      Console.Write(myJagArray[i][j] + " ");
    }
    Console.WriteLine();
  }
  Console.WriteLine();
}
```

Ниже показан вывод, полученный в результате вызова методов `RectMultidimensionalArray()` и `JaggedMultidimensionalArray()`:

```
=> Rectangular multidimensional array:
0       0       0       0
0       1       2       3
0       2       4       6
=> Jagged multidimensional array:
0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

Использование массивов в качестве аргументов и возвращаемых значений

После создания массив можно передавать как аргумент или получать его в виде возвращаемого значения. Например, приведенный ниже метод `PrintArray()` принимает входной массив значений `int` и выводит все его элементы на консоль, а метод `GetStringArray()` заполняет массив значений `string` и возвращает его вызывающему коду:

```
static void PrintArray(int[] myInts)
{
  for(int i = 0; i < myInts.Length; i++)
  {
    Console.WriteLine("Item {0} is {1}", i, myInts[i]);
  }
}
static string[] GetStringArray()
{
  string[] theStrings = {"Hello", "from", "GetStringArray"};
  return theStrings;
}
```

Указанные методы вызываются вполне ожидаемо:

```
static void PassAndReceiveArrays()
{
  Console.WriteLine("=> Arrays as params and return values.");
  // Передать массив в качестве параметра.
  int[] ages = {20, 22, 23, 0} ;
  PrintArray(ages);
```

```
  // Получить массив как возвращаемое значение.
  string[] strs = GetStringArray();
  foreach(string s in strs)
    Console.WriteLine(s);

  Console.WriteLine();
}
```

К настоящему моменту вы должны освоить процесс определения, заполнения и исследования содержимого переменной типа массива C#. Для полноты картины давайте проанализируем роль класса `System.Array`.

Использование базового класса `System.Array`

Каждый создаваемый массив получает значительную часть своей функциональности от класса `System.Array`. Общие члены этого класса позволяют работать с массивом, применяя согласованную объектную модель. В табл. 4.1 приведено краткое описание наиболее интересных членов класса `System.Array` (полное описание всех его членов можно найти в документации).

Таблица 4.1. Избранные члены класса `System.Array`

Член класса `Array`	Описание
`Clear()`	Этот статический метод устанавливает для заданного диапазона элементов в массиве пустые значения (0 для чисел, `null` для объектных ссылок и `false` для булевских значений)
`CopyTo()`	Этот метод используется для копирования элементов из исходного массива в целевой массив
`Length`	Это свойство возвращает количество элементов в массиве
`Rank`	Это свойство возвращает количество измерений в массиве
`Reverse()`	Этот статический метод обращает содержимое одномерного массива
`Sort()`	Этот статический метод сортирует одномерный массив внутренних типов. Если элементы в массиве реализуют интерфейс `IComparer`, то можно сортировать также и специальные типы (см. главы 8 и 10)

Давайте посмотрим на некоторые из членов в действии. Показанный далее вспомогательный метод использует статические методы `Reverse()` и `Clear()` для вывода на консоль информации о массиве строковых типов:

```
static void SystemArrayFunctionality()
{
  Console.WriteLine("=> Working with System.Array.");
  // Инициализировать элементы при запуске.
  string[] gothicBands = {"Tones on Tail", "Bauhaus", "Sisters of Mercy"};

  // Вывести имена в порядке их объявления.
  Console.WriteLine("-> Here is the array:");
  for (int i = 0; i < gothicBands.Length; i++)
  {
    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
  }
```

```
  Console.WriteLine("\n");

  // Обратить порядок следования элементов...
  Array.Reverse(gothicBands);
  Console.WriteLine("-> The reversed array");

  // ...и вывести их.
  for (int i = 0; i < gothicBands.Length; i++)
  {
    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
  }
  Console.WriteLine("\n");

  // Удалить все элементы кроме первого.
  Console.WriteLine("-> Cleared out all but one...");
  Array.Clear(gothicBands, 1, 2);

  for (int i = 0; i < gothicBands.Length; i++)
  {
    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
  }
  Console.WriteLine();
}
```

Вызов метода `SystemArrayFunctionality()` дает в результате следующий вывод:

```
=> Working with System.Array.
-> Here is the array:
Tones on Tail, Bauhaus, Sisters of Mercy,

-> The reversed array
Sisters of Mercy, Bauhaus, Tones on Tail,

-> Cleared out all but one...
Sisters of Mercy, , ,
```

Обратите внимание, что многие члены класса `System.Array` определены как статические и потому вызываются на уровне класса (примерами могут служить методы `Array.Sort()` и `Array.Reverse()`). Методам подобного рода передается массив, подлежащий обработке. Другие члены `System.Array` (такие как свойство `Length`) действуют на уровне объекта, поэтому могут вызываться прямо на типе массива.

Использование индексов и диапазонов (нововведение в версии 8.0)

Для упрощения работы с последовательностями (включая массивы) в версии C# 8 были введены два новых типа и две новых операции, применяемые при работе с массивами:

- `System.Index` представляет индекс в последовательности;
- `System.Range` представляет поддиапазон индексов;
- операция конца (^) указывает, что индекс отсчитывается относительно конца последовательности;
- операция диапазона (`..`) устанавливает в своих операндах начало и конец диапазона.

На заметку! Индексы и диапазоны можно использовать с массивами, строками, Span<T> и ReadOnlySpan<T>.

Как вы уже видели, индексация массивов начинается с нуля (0). Конец последовательности — это длина последовательности минус единица. Показанный выше цикл for, который выводил содержимое массива gothicBands, можно записать по-другому:

```
for (int i = 0; i < gothicBands.Length; i++)
{
  Index idx = i;
  // Вывести имя.
  Console.Write(gothicBands[idx] + ", ");
}
```

Индекс с операцией конца позволяет указывать количество позиций, которые необходимо отсчитать от конца последовательности, начиная с длины. Не забывайте, что последний элемент в последовательности находится в позиции, на единицу меньше длины последовательности, поэтому ^0 приведет к ошибке. В следующем коде элементы массива выводятся в обратном порядке:

```
for (int i = 1; i <= gothicBands.Length; i++)
{
  Index idx = ^i;
  // Вывести имя.
  Console.Write(gothicBands[idx] + ", ");
}
```

Операция диапазона определяет начальный и конечный индексы и обеспечивает доступ к подпоследовательности внутри списка. Начало диапазона является включающим, а конец — исключающим. Например, чтобы извлечь первые два элемента массива, создайте диапазон от 0 (позиция первого элемента) до 2 (на единицу больше желаемой позиции):

```
foreach (var itm in gothicBands[0..2])
{
  // Вывести имя.
  Console.Write(itm + ", ");
}
Console.WriteLine("\n");
```

Диапазоны можно передавать последовательности также с использованием нового типа данных Range, как показано ниже:

```
Range r = 0..2;  // Конец диапазона является исключающим.
foreach (var itm in gothicBands[r])
{
  // Вывести имя.
  Console.Write(itm + ", ");
}
Console.WriteLine("\n");
```

Диапазоны можно определять с применением целых чисел или переменных типа Index. Тот же самый результат будет получен посредством следующего кода:

```
Index idx1 = 0;
```

```
Index idx2 = 2;
Range r = idx1..idx2;  // Конец диапазона является исключающим.
foreach (var itm in gothicBands[r])
{
  // Вывести имя.
  Console.Write(itm + ", ");
}
Console.WriteLine("\n");
```

Если не указано начало диапазона, тогда используется начало последовательности. Если не указан конец диапазона, тогда применяется длина диапазона. Ошибка не возникает, т.к. конец диапазона является исключающим. В предыдущем примере с массивом, содержащим три элемента, все диапазоны представляют одно и то же подмножество:

```
gothicBands[..]
gothicBands[0..^0]
gothicBands[0..3]
```

Понятие методов

Давайте займемся исследованием деталей определения методов. Методы определяются модификатором доступа и возвращаемым типом (или `void`, если ничего не возвращается) и могут принимать параметры или не принимать их. Метод, который возвращает значение вызывающему коду, обычно называется *функцией*, а метод, не возвращающий значение, как правило, называют собственно *методом*.

На заметку! Модификаторы доступа для методов (и классов) раскрываются в главе 5. Параметры методов рассматриваются в следующем разделе.

До настоящего момента в книге каждый из рассматриваемых методов следовал такому базовому формату:

```
// Вспомните, что статические методы могут вызываться
// напрямую без создания экземпляра класса.
class Program
{
  // static возвращаемыйТип ИмяМетода(список параметров)
  //   { /* Реализация */ }
  static int Add(int x, int y){ return x + y; }
}
```

В нескольких последующих главах вы увидите, что методы могут быть реализованы внутри области видимости классов, структур или интерфейсов (нововведение версии C# 8).

Члены, сжатые до выражений

Вы уже знаете о простых методах, возвращающих значения, вроде метода `Add()`. В версии C# 6 появились члены, сжатые до выражений, которые сокращают синтаксис написания однострочных методов. Например, вот как можно переписать метод `Add()`:

```
static int Add(int x, int y) => x + y;
```

Обычно такой прием называют *"синтаксическим сахаром"*, имея в виду, что генерируемый код IL не изменяется по сравнению с первоначальной версией метода. Он является всего лишь другим способом написания метода. Одни находят его более легким для восприятия, другие — нет, так что выбор стиля зависит от ваших персональных предпочтений (или предпочтений команды разработчиков).

На заметку! Не пугайтесь операции =>. Это лямбда-операция, которая подробно рассматривается в главе 12, где также объясняется, каким образом работают члены, сжатые до выражений. Пока просто считайте их сокращением при написании однострочных операторов.

Локальные функции (нововведение в версии 7.0, обновление в версии 9.0)

В версии C# 7.0 появилась возможность создавать методы внутри методов, которые официально называются *локальными функциями*. Локальная функция является функцией, объявленной внутри другой функции, она обязана быть закрытой, в версии C# 8.0 может быть статической (как демонстрируется в следующем разделе) и не поддерживает перегрузку. Локальные функции допускают вложение: внутри одной локальной функции может быть объявлена еще одна локальная функция.

Чтобы взглянуть на средство локальных функций в действии, создайте новый проект консольного приложения по имени `FunWithLocalFunctions`. Предположим, что вы хотите расширить используемый ранее пример с методом `Add()` для включения проверки достоверности входных данных. Задачу можно решить многими способами, простейший из которых предусматривает добавление логики проверки достоверности прямо в сам метод `Add()`. Модифицируйте предыдущий пример следующим образом (логика проверки достоверности представлена комментарием):

```
static int Add(int x, int y)
{
  // Здесь должна выполняться какая-то проверка достоверности.
  return x + y;
}
```

Как видите, крупных изменений здесь нет. Есть только комментарий, в котором указано, что реальный код должен что-то делать. А что, если вы хотите отделить фактическую реализацию цели метода (возвращение суммы аргументов) от логики проверки достоверности аргументов? Вы могли бы создать дополнительные методы и вызывать их из метода `Add()`. Но это потребовало бы создания еще одного метода для использования только в методе `Add()`. Такое решение может оказаться излишеством. Локальные функции позволяют сначала выполнять проверку достоверности и затем инкапсулировать реальную цель метода, определенного внутри метода `AddWrapper()`:

```
static int AddWrapper(int x, int y)
{
  // Здесь должна выполняться какая-то проверка достоверности.
  return Add();
  int Add()
  {
    return x + y;
  }
}
```

Содержащийся в `AddWrapper()` метод `Add()` можно вызывать лишь из объемлющего метода `AddWrapper()`. Почти наверняка вас интересует, что это вам дало? В приведенном примере мало что (если вообще что-либо). Но если функцию `Add()` нужно вызывать во многих местах метода `AddWrapper()`? И вот теперь вы должны осознать, что наличие локальной функции, не видимой за пределами того места, где она необходима, содействует повторному использованию кода. Вы увидите еще больше преимуществ, обеспечиваемых локальными функциями, когда мы будем рассматривать специальные итераторные методы (в главе 8) и асинхронные методы (в главе 15).

На заметку! `AddWrapper()` является примером локальной функции с вложенной локальной функцией. Вспомните, что функции, объявляемые в операторах верхнего уровня, создаются как локальные функции. Локальная функция `Add()` находится внутри локальной функции `AddWrapper()`. Такая возможность обычно не применяется за рамками учебных примеров, но если вам когда-нибудь понадобятся вложенные локальные функции, то вы знаете, что они поддерживаются в C#.

В версии C# 9.0 локальные функции обновлены, чтобы позволить добавлять атрибуты к самой локальной функции, ее параметрам и параметрам типов, как показано далее в примере (не беспокойтесь об атрибуте `NotNullWhen`, который будет раскрыт позже в главе):

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
  foreach (var line in lines)
  {
    if (IsValid(line))
    {
      // Логика обработки...
    }
  }
  bool IsValid([NotNullWhen(true)] string? line)
  {
    return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
  }
}
```

Статические локальные функции (нововведение в версии 8.0)

В версии C# 8 средство локальных функций было усовершенствовано — появилась возможность объявлять локальную функцию как статическую. В предыдущем примере внутри локальной функции `Add()` производилась прямая ссылка на переменные из главной функции. Результатом могут стать неожиданные побочные эффекты, поскольку локальная функция способна изменять значения этих переменных.

Чтобы увидеть возможные побочные эффекты в действии, создайте новый метод по имени `AddWrapperWithSideEffect()` с таким кодом:

```
static int AddWrapperWithSideEffect(int x, int y)
{
  // Здесь должна выполняться какая-то проверка достоверности.
  return Add();

  int Add()
  {
    x += 1;
    return x + y;
  }
}
```

Конечно, приведенный пример настолько прост, что вряд ли что-то подобное встретится в реальном коде. Для предотвращения ошибки подобного рода добавьте к локальной функции модификатор `static`. Это не позволит локальной функции получать прямой доступ к переменным родительского метода, генерируя на этапе компиляции исключение CS8421, "A static local function cannot contain a reference to '<имя переменной>'" (Статическая локальная функция не может содержать ссылку на '<имя переменной>').

Ниже показана усовершенствованная версия предыдущего метода:

```
static int AddWrapperWithStatic(int x, int y)
{
  // Здесь должна выполняться какая-то проверка достоверности.
  return Add(x,y);

  static int Add(int x, int y)
  {
    return x + y;
  }
}
```

Понятие параметров методов

Параметры методов применяются для передачи данных вызову метода. В последующих разделах вы узнаете детали того, как методы (и вызывающий их код) обрабатывают параметры.

Модификаторы параметров для методов

Стандартный способ передачи параметра в функцию — *по значению*. Попросту говоря, если вы не помечаете аргумент каким-то модификатором параметра, тогда в функцию передается копия данных. Как объясняется далее в главе, то, *что* в точности копируется, будет зависеть от того, относится параметр к типу значения или к ссылочному типу.

Хотя определение метода в C# выглядит достаточно понятно, с помощью модификаторов, описанных в табл. 4.2, можно управлять способом передачи аргументов методу.

Чтобы проиллюстрировать использование перечисленных ключевых слов, создайте новый проект консольного приложения по имени `FunWithMethods`. А теперь давайте рассмотрим их роль.

Таблица 4.2. Модификаторы параметров в C#

Модификатор параметра	Описание
(отсутствует)	Если параметр типа значения не помечен модификатором, то предполагается, что он должен передаваться по значению, т.е. вызываемый метод получает копию исходных данных. Параметры ссылочных типов без какого-либо модификатора передаются по ссылке
`out`	Выходным параметрам должны присваиваться значения внутри вызываемого метода, следовательно, они передаются по ссылке. Если в вызываемом методе выходным параметрам не были присвоены значения, тогда компилятор сообщит об ошибке
`ref`	Значение первоначально присваивается в вызывающем коде и может быть необязательно изменено в вызываемом методе (поскольку данные также передаются по ссылке). Если в вызываемом методе параметру `ref` значение не присваивалось, то никакой ошибки компилятор не генерирует
`in`	Появившийся в версии C# 7.2 модификатор `in` указывает на то, что параметр `ref` доступен вызывающему методу только для чтения
`params`	Этот модификатор позволяет передавать переменное количество аргументов в виде единственного логического параметра. Метод может иметь только один модификатор `params`, которым должен быть помечен последний параметр метода. В реальности потребность в использовании модификатора `params` возникает не особенно часто, однако имейте в виду, что он применяется многочисленными методами внутри библиотек базовых классов

Стандартное поведение передачи параметров

Когда параметр не имеет модификатора, поведение для типов значений предусматривает передачу параметра по значению, а для ссылочных типов — по ссылке.

На заметку! Типы значений и ссылочные типы рассматриваются позже в главе.

Стандартное поведение для типов значений

По умолчанию параметр типа значения передается функции *по значению*. Другими словами, если параметр не помечен каким-либо модификатором, то в функцию передается копия данных. Добавьте в класс `Program` следующий метод, который оперирует с двумя параметрами числового типа, передаваемыми по значению:

```
// По умолчанию аргументы типов значений передаются по значению.
static int Add(int x, int y)
{
  int ans = x + y;
  // Вызывающий код не увидит эти изменения,
  // т.к. модифицируется копия исходных данных.
  x = 10000;
  y = 88888;
  return ans;
}
```

Числовые данные относятся к категории *типов значений*. Следовательно, в случае изменения значений параметров внутри контекста члена вызывающий код будет оставаться в полном неведении об этом, потому что изменения вносятся только в *копию* первоначальных данных из вызывающего кода:

```
Console.WriteLine("***** Fun with Methods *****\n");
// Передать две переменные по значению.
int x = 9, y = 10;
Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
                  // Значения перед вызовом
Console.WriteLine("Answer is: {0}", Add(x, y));
                  // Результат сложения
Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
                  // Значения после вызова
Console.ReadLine();
```

Как видно в показанном далее выводе, значения x и y вполне ожидаемо остаются идентичными до и после вызова метода Add(),поскольку элементы данных передавались по значению. Таким образом, любые изменения параметров, производимые внутри метода Add(), вызывающему коду не видны, т.к. метод Add() оперирует на копии данных.

```
***** Fun with Methods *****
Before call: X: 9, Y: 10
Answer is: 19
After call: X: 9, Y: 10
```

Стандартное поведение для ссылочных типов

Стандартный способ, которым параметр ссылочного типа отправляется функции, предусматривает *передачу по ссылке для его свойств, но передачу по значению для него самого*. Детали будут представлены позже в главе после объяснения типов значений и ссылочных типов.

На заметку! Несмотря на то что строковый тип данных формально относится к ссылочным типам, как обсуждалось в главе 3, он является особым случаем. Когда строковый параметр не имеет какого-либо модификатора, он передается по значению.

Использование модификатора out (обновление в версии 7.0)

Теперь мы рассмотрим *выходные параметры*. Перед покиданием области видимости метода, который был определен для приема выходных параметров (посредством ключевого слова out), им должны присваиваться соответствующие значения (иначе компилятор сообщит об ошибке). В целях демонстрации ниже приведена альтернативная версия метода AddUsingOutParam(), которая возвращает сумму двух целых чисел с применением модификатора out (обратите внимание, что возвращаемым значением метода теперь является void):

```
// Значения выходных параметров должны быть
// установлены внутри вызываемого метода.
static void  AddUsingOutParam(int x, int y, out int ans)
{
  ans = x + y;
}
```

Вызов метода с выходными параметрами также требует использования модификатора `out`. Однако предварительно устанавливать значения локальных переменных, которые передаются в качестве выходных параметров, вовсе не обязательно (после вызова эти значения все равно будут утрачены). Причина, по которой компилятор позволяет передавать на первый взгляд неинициализированные данные, связана с тем, что вызываемый метод *обязан* выполнить присваивание. Чтобы вызвать обновленный метод `AddUsingOutParam()`, создайте переменную типа `int` и примените в вызове модификатор `out`:

```
int ans;
AddUsingOutParam(90, 90, out ans);
```

Начиная с версии C# 7.0, больше нет нужды объявлять параметры `out` до их применения. Другими словами, они могут объявляться внутри вызова метода:

```
AddUsingOutParam(90, 90, out int ans);
```

В следующем коде представлен пример вызова метода с встраиваемым объявлением параметра `out`:

```
Console.WriteLine("***** Fun with Methods *****");
// Присваивать начальные значения локальным переменным, используемым
// как выходные параметры, не обязательно при условии, что они
// применяются в таком качестве впервые.
// Версия C# 7 позволяет объявлять параметры out в вызове метода.
AddUsingOutParam(90, 90, out int ans);
Console.WriteLine("90 + 90 = {0}", ans);
Console.ReadLine();
```

Предыдущий пример по своей природе предназначен только для иллюстрации; на самом деле нет никаких причин возвращать значение суммы через выходной параметр. Тем не менее, модификатор `out` в C# служит действительно практичной цели: он позволяет вызывающему коду получать несколько выходных значений из единственного вызова метода:

```
// Возвращение множества выходных параметров.
static void FillTheseValues(out int a, out string b, out bool c)
{
  a = 9;
  b = "Enjoy your string.";
  c = true;
}
```

Теперь вызывающий код имеет возможность обращаться к методу `FillTheseValues()`. Не забывайте, что модификатор `out` должен применяться как при вызове, так и при реализации метода:

```
Console.WriteLine("***** Fun with Methods *****");
FillTheseValues(out int i, out string str, out bool b);
Console.WriteLine("Int is: {0}", i);  // Вывод целочисленного значения
Console.WriteLine("String is: {0}", str); // Вывод строкового значения
Console.WriteLine("Boolean is: {0}", b);  // Вывод булевского значения
Console.ReadLine();
```

На заметку! В версии C# 7 также появились кортежи, представляющие собой еще один способ возвращения множества значений из вызова метода. Они будут описаны далее в главе.

Всегда помните о том, что перед выходом из области видимости метода, определяющего выходные параметры, этим параметрам *должны* быть присвоены допустимые значения. Таким образом, следующий код вызовет ошибку на этапе компиляции, потому что внутри метода отсутствует присваивание значения выходному параметру:

```
static void ThisWontCompile(out int a)
{
  Console.WriteLine("Error! Forgot to assign output arg!");
            // Ошибка! Забыли присвоить значение выходному параметру!
}
```

Отбрасывание параметров `out` (нововведение в версии 7.0)

Если значение параметра `out` не интересует, тогда в качестве заполнителя можно использовать отбрасывание. Отбрасывания представляют собой временные фиктивные переменные, которые намеренно не используются. Их присваивание не производится, они не имеют значения и для них может вообще не выделяться память. Отбрасывание способно обеспечить выигрыш в производительности, а также сделать код более читабельным. Его можно применять с параметрами `out`, кортежами (как объясняется позже в главе), сопоставлением с образцом (см. главы 6 и 8) или даже в качестве автономных переменных.

Например, если вы хотите получить значение для `int` в предыдущем примере, но остальные два параметра вас не волнуют, тогда можете написать такой код:

```
// Здесь будет получено значение только для a;
// значения для других двух параметров игнорируются.
FillTheseValues(out int a, out _, out _);
```

Обратите внимание, что вызываемый метод по-прежнему выполняет работу, связанную с установкой значений для всех трех параметров; просто последние два параметра *отбрасываются*, когда происходит возврат из вызова метода.

Модификатор `out` в конструкторах и инициализаторах (нововведение в версии 7.3)

В версии C# 7.3 были расширены допустимые местоположения для использования параметра `out`. В дополнение к методам параметры конструкторов, инициализаторы полей и свойств, а также конструкции запросов можно декорировать модификатором `out`. Примеры будут представлены позже в главе.

Использование модификатора `ref`

А теперь посмотрим, как в C# используется модификатор `ref`. Ссылочные параметры необходимы, когда вы хотите разрешить методу манипулировать различными элементами данных (и обычно изменять их значения), которые объявлены в вызывающем коде, таком как процедура сортировки или обмена.

Обратите внимание на отличия между ссылочными и выходными параметрами.

- Выходные параметры не нуждаются в инициализации перед передачей методу. Причина в том, что метод до своего завершения обязан самостоятельно присваивать значения выходным параметрам.
- Ссылочные параметры должны быть инициализированы перед передачей методу. Причина связана с передачей ссылок на существующие переменные. Если начальные значения им не присвоены, то это будет равнозначно работе с неинициализированными локальными переменными.

Давайте рассмотрим применение ключевого слова `ref` на примере метода, меняющего местами значения двух переменных типа `string` (естественно, здесь мог бы использоваться любой тип данных, включая `int`, `bool`, `float` и т.д.):

```
// Ссылочные параметры.
public static void SwapStrings(ref string s1, ref string s2)
{
  string tempStr = s1;
  s1 = s2;
  s2 = tempStr;
}
```

Метод `SwapStrings()` можно вызвать следующим образом:

```
Console.WriteLine("***** Fun with Methods *****");
string str1 = "Flip";
string str2 = "Flop";
Console.WriteLine("Before: {0}, {1} ", str1, str2);  // До
SwapStrings(ref str1, ref str2);
Console.WriteLine("After: {0}, {1} ", str1, str2);   // После
Console.ReadLine();
```

Здесь вызывающий код присваивает начальные значения локальным строковым данным (`str1` и `str2`). После вызова метода `SwapStrings()` строка `str1` будет содержать значение `"Flop"`, а строка `str2` — значение `"Flip"`:

```
Before: Flip, Flop
After: Flop, Flip
```

Использование модификатора `in` (нововведение в версии 7.2)

Модификатор `in` обеспечивает передачу значения по ссылке (для типов значений и ссылочных типов) и препятствует модификации значений в вызываемом методе. Это четко формулирует проектный замысел в коде, а также потенциально снижает нагрузку на память. Когда параметры типов значений передаются по значению, они (внутренне) копируются вызываемым методом. Если объект является большим (вроде крупной структуры), тогда добавочные накладные расходы на создание копии для локального использования могут оказаться значительными. Кроме того, даже когда параметры ссылочных типов передаются без модификатора, в вызываемом методе их можно модифицировать. Обе проблемы решаются с применением модификатора `in`.

В рассмотренном ранее методе `Add()` есть две строки кода, которые изменяют параметры, но не влияют на значения для вызывающего метода. Влияние на значения отсутствует из-за того, что метод `Add()` создает копию переменных `x` и `y` с целью локального использования. Пока вызывающий метод не имеет неблагоприятных побочных эффектов, но что произойдет, если бы код метода `Add()` был таким, как показано ниже?

```
static int Add2(int x,int y)
{
  x = 10000;
  y = 88888;
  int ans = x + y;
  return ans;
}
```

В данном случае метод возвращает значение 98888 независимо от переданных ему чисел, что очевидно представляет собой проблему. Чтобы устранить ее, код метода понадобится изменить следующим образом:

```
static int AddReadOnly(in int x,in int y)
{
  // Ошибка CS8331 Cannot assign to variable 'in int'
  // because it is a readonly variable
  // Не удается присвоить значение переменной in int,
  // поскольку она допускает только чтение
  // x = 10000;
  // y = 88888;
  int ans = x + y;
  return ans;
}
```

Когда в коде предпринимается попытка изменить значения параметров, компилятор сообщит об ошибке CS8331, указывая на то, что значения не могут быть изменены из-за наличия модификатора in.

Использование модификатора params

В языке C# поддерживаются *массивы параметров* с использованием ключевого слова params, которое позволяет передавать методу переменное количество идентично типизированных параметров (или классов, связанных отношением наследования) в виде *единственного логического параметра*. Вдобавок аргументы, помеченные ключевым словом params, могут обрабатываться, когда вызывающий код передает строго типизированный массив или список элементов, разделенных запятыми. Да, это может сбивать с толку! В целях прояснения предположим, что вы хотите создать функцию, которая позволяет вызывающему коду передавать любое количество аргументов и возвращает их среднее значение.

Если вы прототипируете данный метод так, чтобы он принимал массив значений double, тогда в вызывающем коде придется сначала определить массив, затем заполнить его значениями и, наконец, передать его методу. Однако если вы определите метод CalculateAverage() как принимающий параметр params типа double[], то вызывающий код может просто передавать список значений double, разделенных запятыми. "За кулисами" список значений double будет упакован в массив типа double.

```
// Возвращение среднего из некоторого количества значений double.
static double CalculateAverage(params double[] values)
{
  Console.WriteLine("You sent me {0} doubles.", values.Length);
  double sum = 0;
```

```
  if(values.Length == 0)
  {
    return sum;
  }
  for (int i = 0; i < values.Length; i++)
  {
    sum += values[i];
  }
  return (sum / values.Length);
}
```

Метод `CalculateAverage()` был определен для приема массива параметров типа `double`. Фактически он ожидает передачи любого количества (включая ноль) значений `double` и вычисляет их среднее. Метод может вызываться любым из показанных далее способов:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  // Передать список значений double, разделенных запятыми...
  double average;
  average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
  // Вывод среднего значения для переданных данных
  Console.WriteLine("Average of data is: {0}", average);
  // ...или передать массив значений double.
  double[] data = { 4.0, 3.2, 5.7 };
  average = CalculateAverage(data);
  // Вывод среднего значения для переданных данных
  Console.WriteLine("Average of data is: {0}", average);
  // Среднее из 0 равно 0!
  // Вывод среднего значения для переданных данных
  Console.WriteLine("Average of data is: {0}", CalculateAverage());
  Console.ReadLine();
}
```

Если модификатор `params` в определении метода `CalculateAverage()` не задействован, тогда его первый вызов приведет к ошибке на этапе компиляции, т.к. компилятору не удастся найти версию `CalculateAverage()`, принимающую пять аргументов типа `double`.

На заметку! Во избежание любой неоднозначности язык C# требует, чтобы метод поддерживал только один параметр `params`, который должен быть последним в списке параметров.

Как и можно было догадаться, данный прием — всего лишь удобство для вызывающего кода, потому что .NET Core Runtime создает массив по мере необходимости. В момент, когда массив окажется внутри области видимости вызываемого метода, его можно трактовать как полноценный массив .NET Core, обладающий всей функциональностью базового библиотечного класса `System.Array`. Взгляните на вывод:

```
You sent me 5 doubles.
Average of data is: 32.864
You sent me 3 doubles.
Average of data is: 4.3
You sent me 0 doubles.
Average of data is: 0
```

Определение необязательных параметров

Язык C# дает возможность создавать методы, которые могут принимать *необязательные аргументы*. Такой прием позволяет вызывать метод, опуская ненужные аргументы, при условии, что подходят указанные для них стандартные значения.

Для иллюстрации работы с необязательными аргументами предположим, что имеется метод по имени `EnterLogData()` с одним необязательным параметром:

```
static void EnterLogData(string message, string owner = "Programmer")
{
  Console.Beep();
  Console.WriteLine("Error: {0}", message);       // Сведения об ошибке
  Console.WriteLine("Owner of Error: {0}", owner); // Владелец ошибки
}
```

Здесь последнему аргументу `string` было присвоено стандартное значение `"Programmer"` через операцию присваивания внутри определения параметров. В результате метод `EnterLogData()` можно вызывать двумя способами:

```
Console.WriteLine("***** Fun with Methods *****");
...
EnterLogData("Oh no! Grid can't find data");
EnterLogData("Oh no! I can't find the payroll data", "CFO");
Console.ReadLine();
```

Из-за того, что в первом вызове `EnterLogData()` не был указан второй аргумент `string`, будет использоваться его стандартное значение — `"Programmer"`. Во втором вызове `EnterLogData()` для второго аргумента передано значение `"CFO"`.

Важно понимать, что значение, присваиваемое необязательному параметру, должно быть известно на этапе компиляции и не может вычисляться во время выполнения (если вы попытаетесь сделать это, то компилятор сообщит об ошибке). В целях иллюстрации модифицируйте метод `EnterLogData()`, добавив к нему дополнительный необязательный параметр:

```
// Ошибка! Стандартное значение для необязательного
// аргумента должно быть известно на этапе компиляции!
static void EnterLogData(string message,
       string owner = "Programmer", DateTime timeStamp = DateTime.Now)
{
  Console.Beep();
  Console.WriteLine("Error: {0}", message);       // Сведения об ошибке
  Console.WriteLine("Owner of Error: {0}", owner); // Владелец ошибки
  Console.WriteLine("Time of Error: {0}", timeStamp);
                  // Время возникновения ошибки
}
```

Такой код не скомпилируется, поскольку значение свойства `Now` класса `DateTime` вычисляется во время выполнения, а не на этапе компиляции.

На заметку! Во избежание неоднозначности необязательные параметры должны всегда помещаться в конец сигнатуры метода. Если необязательные параметры обнаруживаются перед обязательными, тогда компилятор сообщит об ошибке.

Использование именованных параметров (обновление в версии 7.2)

Еще одним полезным языковым средством C# является поддержка *именованных аргументов*. Именованные аргументы позволяют вызывать метод с указанием значений параметров в любом желаемом порядке. Таким образом, вместо передачи параметров исключительно по позициям (как делается в большинстве случаев) можно указывать имя каждого аргумента, двоеточие и конкретное значение. Чтобы продемонстрировать использование именованных аргументов, добавьте в класс `Program` следующий метод:

```
static void DisplayFancyMessage(ConsoleColor textColor,
  ConsoleColor backgroundColor, string message)
{
  //Сохранить старые цвета для их восстановления после вывода сообщения
  ConsoleColor oldTextColor = Console.ForegroundColor;
  ConsoleColor oldbackgroundColor = Console.BackgroundColor;

  // Установить новые цвета и вывести сообщение.
  Console.ForegroundColor = textColor;
  Console.BackgroundColor = backgroundColor;
  Console.WriteLine(message);

  // Восстановить предыдущие цвета.
  Console.ForegroundColor = oldTextColor;
  Console.BackgroundColor = oldbackgroundColor;
}
```

Теперь, когда метод `DisplayFancyMessage()` написан, можно было бы ожидать, что при его вызове будут передаваться две переменные типа `ConsoleColor`, за которыми следует переменная типа `string`. Однако с помощью именованных аргументов метод `DisplayFancyMessage()` допустимо вызывать и так, как показано ниже:

```
Console.WriteLine("***** Fun with Methods *****");

DisplayFancyMessage(message: "Wow! Very Fancy indeed!",
  textColor: ConsoleColor.DarkRed,
  backgroundColor: ConsoleColor.White);
DisplayFancyMessage(backgroundColor: ConsoleColor.Green,
  message: "Testing...",
  textColor: ConsoleColor.DarkBlue);
Console.ReadLine();
```

В версии C# 7.2 правила применения именованных аргументов слегка изменились. До выхода C# 7.2 при вызове метода позиционные параметры должны были располагаться перед любыми именованными параметрами. В C# 7.2 и последующих версиях именованные и неименованные параметры можно смешивать, если параметры находятся в корректных позициях.

На заметку! Хотя в C# 7.2 и последующих версиях именованные и позиционные аргументы можно смешивать, поступать так — не особо удачная идея. Возможность не значит обязательность!

Ниже приведен пример:

```
// Все нормально, т.к. позиционные аргументы находятся перед именованными.
DisplayFancyMessage(ConsoleColor.Blue,
  message: "Testing...",
  backgroundColor: ConsoleColor.White);

// Все нормально, т.к. все аргументы располагаются в корректном порядке.
DisplayFancyMessage(textColor: ConsoleColor.White,
                    backgroundColor:ConsoleColor.Blue,
                    "Testing...");

// ОШИБКА в вызове, поскольку позиционные аргументы следуют
// после именованных.
DisplayFancyMessage(message: "Testing...",
  backgroundColor: ConsoleColor.White,
  ConsoleColor.Blue);
```

Даже если оставить в стороне указанное ограничение, то все равно может возникать вопрос: при каких условиях вообще требуется такая языковая конструкция? В конце концов, для чего нужно менять позиции аргументов метода?

Как выясняется, при наличии метода, в котором определены необязательные аргументы, данное средство может оказаться по-настоящему полезным. Предположим, что метод `DisplayFancyMessage()` переписан с целью поддержки необязательных аргументов, для которых указаны подходящие стандартные значения:

```
static void DisplayFancyMessage(
  ConsoleColor textColor = ConsoleColor.Blue,
  ConsoleColor backgroundColor = ConsoleColor.White,
  string message = "Test Message")
{
  ...
}
```

Учитывая, что каждый аргумент имеет стандартное значение, именованные аргументы позволяют указывать в вызывающем коде только те параметры, которые не должны принимать стандартные значения. Следовательно, если нужно, чтобы значение `"Hello!"` появлялось в виде текста синего цвета на белом фоне, то в вызывающем коде можно просто записать так:

```
DisplayFancyMessage(message: "Hello!");
```

Если же необходимо, чтобы строка `"Test Message"` выводилась синим цветом на зеленом фоне, тогда должен применяться такой вызов:

```
DisplayFancyMessage(backgroundColor: ConsoleColor.Green);
```

Как видите, необязательные аргументы и именованные параметры часто работают бок о бок. В завершение темы построения методов C# необходимо ознакомиться с концепцией *перегрузки методов*.

Понятие перегрузки методов

Подобно другим современным языкам объектно-ориентированного программирования в C# разрешена *перегрузка* методов. Выражаясь просто, когда определяется набор идентично именованных методов, которые отличаются друг от друга количеством (или типами) параметров, то говорят, что такой метод был *перегружен*.

Чтобы оценить удобство перегрузки методов, давайте представим себя на месте разработчика, использующего Visual Basic 6.0 (VB6). Предположим, что на языке VB6 создается набор методов, возвращающих сумму значений разнообразных типов (`Integer`, `Double` и т.д.). С учетом того, что VB6 не поддерживает перегрузку методов, придется определить уникальный набор методов, каждый из которых будет делать по существу одно и то же (возвращать сумму значений аргументов):

```
' Примеры кода VB6.
Public Function AddInts(ByVal x As Integer, ByVal y As Integer) As Integer
  AddInts = x + y
End Function

Public Function AddDoubles(ByVal x As Double, ByVal y As Double) As Double
  AddDoubles = x + y
End Function

Public Function AddLongs(ByVal x As Long, ByVal y As Long) As Long
  AddLongs = x + y
End Function
```

Такой код не только становится трудным в сопровождении, но и заставляет помнить имена всех методов. Применяя перегрузку, вызывающему коду можно предоставить возможность обращения к единственному методу по имени `Add()`. Ключевой аспект в том, чтобы обеспечить для каждой версии метода отличающийся набор аргументов (различий только в возвращаемом типе не достаточно).

На заметку! Как будет объясняться в главе 10, существует возможность построения обобщенных методов, которые переносят концепцию перегрузки на новый уровень. Используя обобщения, можно определять заполнители типов для реализации метода, которая указывается во время его вызова.

Чтобы попрактиковаться с перегруженными методами, создайте новый проект консольного приложения по имени `FunWithMethodOverloading`. Добавьте новый класс по имени `AddOperations.cs` и приведите его код к следующему виду:

```
namespace FunWithMethodOverloading {
  // Код C#.
  public static class AddOperations
  {
    // Перегруженный метод Add().
    public static int Add(int x, int y)
    {
      return x + y;
    }
    // Перегруженный метод Add().
    public static double Add(double x, double y)
    {
      return x + y;
    }
    // Перегруженный метод Add().
    public static long Add(long x, long y)
    {
      return x + y;
    }
  }
}
```

Замените код в Program.cs показанным ниже кодом:

```
using System;
using FunWithMethodOverloading;
using static FunWithMethodOverloading.AddOperations;

Console.WriteLine("***** Fun with Method Overloading *****\n");

// Вызов версии int метода Add()
Console.WriteLine(Add(10, 10));

// Вызов версии long метода Add() с использованием нового разделителя
групп цифр
Console.WriteLine(Add(900_000_000_000, 900_000_000_000));

// Вызов версии double метода Add()
Console.WriteLine(Add(4.3, 4.4));

Console.ReadLine();
```

На заметку! Оператор using static будет раскрыт в главе 5. Пока считайте его клавиатурным сокращением для использования методов, содержащихся в статическом классе по имени AddOperations из пространства имен FunWithMethodOverloading.

В операторах верхнего уровня вызываются три разных версии метода Add() с применением для каждой отличающегося типа данных.

Среды Visual Studio и Visual Studio Code оказывают помощь при вызове перегруженных методов. Когда вводится имя перегруженного метода (такого как хорошо знакомый метод Console.WriteLine()), средство IntelliSense отображает список всех его доступных версий. Обратите внимание, что по списку можно перемещаться с применением клавиш со стрелками вниз и вверх (рис. 4.1).

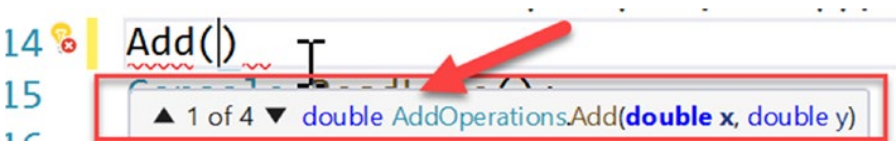

Рис. 4.1. Средство IntelliSense в Visual Studio для перегруженных методов

Если перегруженная версия принимает необязательные параметры, тогда компилятор будет выбирать метод, лучше всего подходящий для вызывающего кода, на основе именованных и/или позиционных аргументов. Добавьте следующий метод:

```
static int Add(int x, int y, int z = 0)
{
  return x + (y*z);
}
```

Если необязательный аргумент в вызывающем коде не передается, то компилятор даст соответствие с первой сигнатурой (без необязательного параметра). Хотя существует набор правил для нахождения методов, обычно имеет смысл избегать создания методов, которые отличаются только необязательными параметрами.

Наконец, in, ref и out не считаются частью сигнатуры при перегрузке методов, когда используется более одного модификатора. Другими словами, приведенные ниже перегруженные версии будут приводить к ошибке на этапе компиляции:

```
static int Add(ref int x) { /* */ }
static int Add(out int x) { /* */ }
```

Однако если модификатор `in`, `ref` или `out` применяется только в одном методе, тогда компилятор способен проводить различие между сигнатурами. Таким образом, следующий код разрешен:

```
static int Add(ref int x) { /* */ }
static int Add(int x) { /* */ }
```

На этом начальное изучение построения методов с использованием синтаксиса C# завершено. Теперь давайте выясним, как строить перечисления и структуры и манипулировать ими.

Понятие типа enum

Вспомните из главы 1, что система типов .NET Core состоит из классов, структур, перечислений, интерфейсов и делегатов. Чтобы начать исследование таких типов, рассмотрим роль *перечисления* (`enum`), создав новый проект консольного приложения по имени `FunWithEnums`.

На заметку! Не путайте термины перечисление и перечислитель; они обозначают совершенно разные концепции. Перечисление — специальный тип данных, состоящих из пар "имя-значение". Перечислитель — тип класса или структуры, который реализует интерфейс .NET Core по имени `IEnumerable`. Обычно упомянутый интерфейс реализуется классами коллекций, а также классом `System.Array`. Как будет показано в главе 8, поддерживающие `IEnumerable` объекты могут работать с циклами `foreach`.

При построении какой-либо системы часто удобно создавать набор символических имен, которые отображаются на известные числовые значения. Например, в случае создания системы начисления заработной платы может возникнуть необходимость в ссылке на типы сотрудников с применением констант вроде `VicePresident` (вице-президент), `Manager` (менеджер), `Contractor` (подрядчик) и `Grunt` (рядовой сотрудник). Для этой цели в C# поддерживается понятие специальных перечислений. Например, далее представлено специальное перечисление по имени `EmpTypeEnum` (его можно определить в том же файле, где находятся операторы верхнего уровня, если определение будет помещено в конец файла):

```
using System;

Console.WriteLine("**** Fun with Enums *****\n");
Console.ReadLine();

// Здесь должны находиться локальные функции:

// Специальное перечисление.
enum EmpTypeEnum
{
  Manager,              // = 0
  Grunt,                // = 1
  Contractor,           // = 2
  VicePresident         // = 3
}
```

На заметку! По соглашению имена типов перечислений обычно снабжаются суффиксом Enum. Поступать так необязательно, но подобный подход улучшает читабельность кода.

В перечислении EmpTypeEnum определены четыре именованные константы, которые соответствуют дискретным числовым значениям. По умолчанию первому элементу присваивается значение 0, а остальным элементам значения устанавливаются по схеме n+1. При желании исходное значение можно изменять подходящим образом. Например, если имеет смысл нумеровать члены EmpTypeEnum со значения 102 до 105, тогда можно поступить следующим образом:

```
// Начать нумерацию со значения 102.
enum EmpTypeEnum
{
  Manager = 102,
  Grunt,                // = 103
  Contractor,           // = 104
  VicePresident         // = 105
}
```

Нумерация в перечислениях не обязана быть последовательной и содержать только уникальные значения. Если (по той или иной причине) перечисление EmpTypeEnum необходимо сконфигурировать так, как показано ниже, то компиляция пройдет гладко и без ошибок:

```
// Значения элементов в перечислении не обязательно должны
// быть последовательными!
enum EmpTypeEnum
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

Управление хранилищем, лежащим в основе перечисления

По умолчанию для хранения значений перечисления используется тип System.Int32 (int в языке C#); тем не менее, при желании его легко заменить. Перечисления в C# можно определять в похожей манере для любых основных системных типов (byte, short, int или long). Например, чтобы значения перечисления EmpTypeEnum хранились с применением типа byte, а не int, можно записать так:

```
// На этот раз для элементов EmpTypeEnum используется тип byte.
enum EmpTypeEnum : byte
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

Изменение типа, лежащего в основе перечисления, может быть полезным при построении приложения .NET Core, которое планируется развертывать на устройствах с небольшим объемом памяти, а потому необходимо экономить память везде, где только

возможно. Конечно, если в качестве типа хранилища для перечисления указан byte, то каждое значение должно входить в диапазон его допустимых значений. Например, следующая версия EmpTypeEnum приведет к ошибке на этапе компиляции, т.к. значение 999 не умещается в диапазон допустимых значений типа byte:

```
// Ошибка на этапе компиляции! Значение 999 слишком велико для типа byte!
enum EmpTypeEnum : byte
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 999
}
```

Объявление переменных типа перечисления

После установки диапазона и типа хранилища перечисление можно использовать вместо так называемых "магических чисел". Поскольку перечисления — всего лишь определяемые пользователем типы данных, их можно применять как возвращаемые значения функций, параметры методов, локальные переменные и т.д. Предположим, что есть метод по имени AskForBonus(), который принимает в качестве единственного параметра переменную EmpTypeEnum. На основе значения входного параметра в окно консоли будет выводиться подходящий ответ на запрос о надбавке к зарплате.

```
Console.WriteLine("**** Fun with Enums *****");
// Создать переменную типа EmpTypeEnum.
EmpTypeEnum emp = EmpTypeEnum.Contractor;
AskForBonus(emp);
Console.ReadLine();

// Перечисления как параметры.
static void AskForBonus(EmpTypeEnum e)
{
  switch (e)
  {
    case EmpType.Manager:
      Console.WriteLine("How about stock options instead?");
                    // Не желаете ли взамен фондовые опционы?
      break;
    case EmpType.Grunt:
      Console.WriteLine("You have got to be kidding...");
                    // Вы должно быть шутите...
      break;
    case EmpType.Contractor:
      Console.WriteLine("You already get enough cash...");
                    // Вы уже получаете вполне достаточно...
      break;
    case EmpType.VicePresident:
      Console.WriteLine("VERY GOOD, Sir!");
                    // Очень хорошо, сэр!
      break;
  }
}
```

Обратите внимание, что когда переменной enum присваивается значение, вы должны указывать перед этим значением (Grunt) имя самого перечисления (EmpTypeEnum). Из-за того, что перечисления представляют собой фиксированные наборы пар "имя-значение", установка переменной enum в значение, которое не определено прямо в перечислимом типе, не допускается:

```
static void ThisMethodWillNotCompile()
{
  // Ошибка! SalesManager отсутствует в перечислении EmpTypeEnum!
  EmpType emp = EmpType.SalesManager;

  // Ошибка! Не указано имя EmpTypeEnum перед значением Grunt!
  emp = Grunt;
}
```

Использование типа System.Enum

С перечислениями .NET Core связан один интересный аспект — они получают свою функциональность от класса System.Enum. В классе System.Enum определено множество методов, которые позволяют исследовать и трансформировать заданное перечисление. Одним из них является метод Enum.GetUnderlyingType(), который возвращает тип данных, используемый для хранения значений перечислимого типа (System.Byte в текущем объявлении EmpTypeEnum):

```
Console.WriteLine("**** Fun with Enums *****");
...
// Вывести тип хранилища для значений перечисления.
Console.WriteLine("EmpTypeEnum uses a {0} for storage",
  Enum.GetUnderlyingType(emp.GetType()));
Console.ReadLine();
```

Метод Enum.GetUnderlyingType() требует передачи System.Type в качестве первого параметра. В главе 15 будет показано, что класс Type представляет описание метаданных для конкретной сущности .NET Core.

Один из возможных способов получения метаданных (как демонстрировалось ранее) предусматривает применение метода GetType(), который является общим для всех типов в библиотеках базовых классов .NET Core. Другой подход заключается в использовании операции typeof языка C#. Преимущество такого способа связано с тем, что он не требует объявления переменной сущности, описание метаданных которой требуется получить:

```
//На этот раз для получения информации о типе используется операция typeof
Console.WriteLine("EmpTypeEnum uses a {0} for storage",
  Enum.GetUnderlyingType(typeof(EmpTypeEnum)));
```

Динамическое обнаружение пар "имя-значение" перечисления

Кроме метода Enum.GetUnderlyingType() все перечисления C# поддерживают метод по имени ToString(), который возвращает строковое имя текущего значения перечисления. Ниже приведен пример:

```
EmpTypeEnum emp = EmpTypeEnum.Contractor;
...
// Выводит строку "emp is a Contractor."
Console.WriteLine("emp is a {0}.", emp.ToString());
Console.ReadLine();
```

Если интересует не имя, а значение заданной переменной перечисления, то можно просто привести ее к лежащему в основе типу хранилища, например:

```
Console.WriteLine("**** Fun with Enums *****");
EmpTypeEnum emp = EmpTypeEnum.Contractor;
...
// Выводит строку "Contractor = 100".
Console.WriteLine("{0} = {1}", emp.ToString(), (byte)emp);
Console.ReadLine();
```

На заметку! Статический метод `Enum.Format()` предлагает более высокий уровень форматирования за счет указания флага желаемого формата. Полный список флагов форматирования ищите в документации.

В типе `System.Enum` определен еще один статический метод по имени `GetValues()`, возвращающий экземпляр класса `System.Array`. Каждый элемент в массиве соответствует члену в указанном перечислении. Рассмотрим следующий метод, который выводит на консоль пары "имя-значение" из перечисления, переданного в качестве параметра:

```
// Этот метод выводит детали любого перечисления.
static void EvaluateEnum(System.Enum e)
{
  Console.WriteLine("=> Information about {0}", e.GetType().Name);
  // Вывести лежащий в основе тип хранилища.
  Console.WriteLine("Underlying storage type: {0}",
    Enum.GetUnderlyingType(e.GetType()));
  // Получить все пары "имя-значение" для входного параметра.
  Array enumData = Enum.GetValues(e.GetType());
  Console.WriteLine("This enum has {0} members.", enumData.Length);
  // Вывести строковое имя и ассоциированное значение,
  // используя флаг формата D (см. главу 3).
  for(int i = 0; i < enumData.Length; i++)
  {
    Console.WriteLine("Name: {0}, Value: {0:D}",
      enumData.GetValue(i));
  }
  Console.WriteLine();
}
```

Чтобы протестировать метод `EvaluateEnum()`, модифицируйте код для создания переменных нескольких типов перечислений, объявленных в пространстве имен `System` (вместе с перечислением `EmpTypeEnum`):

```
Console.WriteLine("**** Fun with Enums *****");
...
EmpTypeEnum e2 = EmpType.Contractor;

// Эти типы являются перечислениями из пространства имен System.
DayOfWeek day = DayOfWeek.Monday;
ConsoleColor cc = ConsoleColor.Gray;

EvaluateEnum(e2);
EvaluateEnum(day);
EvaluateEnum(cc);
Console.ReadLine();
```

Ниже показана часть вывода:

```
=> Information about DayOfWeek
Underlying storage type: System.Int32
This enum has 7 members.
Name: Sunday, Value: 0
Name: Monday, Value: 1
Name: Tuesday, Value: 2
Name: Wednesday, Value: 3
Name: Thursday, Value: 4
Name: Friday, Value: 5
Name: Saturday, Value: 6
```

В ходе чтения книги вы увидите, что перечисления широко применяются во всех библиотеках базовых классов .NET Core. При работе с любым перечислением всегда помните о возможности взаимодействия с парами "имя-значение", используя члены класса `System.Enum`.

Использование перечислений, флагов и побитовых операций

Побитовые операции предлагают быстрый механизм для работы с двоичными числами на уровне битов. В табл. 4.3 представлены побитовые операции C#, описаны их действия и приведены примеры.

Таблица 4.3. Побитовые операции

Операция	Что делает	Пример
& (И)	Копирует бит, если он присутствует в обоих операндах	0110 & 0100 = 0100 (4)
| (ИЛИ)	Копирует бит, если он присутствует в одном из операндов или в обоих	0110 | 0100 = 0110 (6)
^ (исключающее ИЛИ)	Копирует бит, если он присутствует в одном из операндов, но не в обоих	0110 ^ 0100 = 0010 (2)
~ (дополнение до единицы)	Переключает биты	~0110 = –7 (из-за переполнения)
<< (сдвиг влево)	Сдвигает биты влево	0110 << 1 = 1100 (12)
>> (сдвиг вправо)	Сдвигает биты вправо	0110 << 1 = 0011 (3)

Чтобы взглянуть на побитовые операции в действии, создайте новый проект консольного приложения по имени `FunWithBitwiseOperations`. Поместите в файл `Program.cs` следующий код:

```
using System;
using FunWithBitwiseOperations;
Console.WriteLine("===== Fun wih Bitwise Operations");
Console.WriteLine("6 & 4 = {0} | {1}", 6 & 4, Convert.ToString((6 & 4),2));
Console.WriteLine("6 | 4 = {0} | {1}", 6 | 4, Convert.ToString((6 | 4),2));
Console.WriteLine("6 ^ 4 = {0} | {1}", 6 ^ 4, Convert.ToString((6 ^ 4),2));
Console.WriteLine("6 << 1 = {0} | {1}", 6 << 1,Convert.ToString((6 << 1),2));
Console.WriteLine("6 >> 1 = {0} | {1}", 6 >> 1,Convert.ToString((6 >> 1),2));
Console.WriteLine("~6 = {0} | {1}", ~6, Convert.ToString(~((short)6),2));
Console.WriteLine("Int.MaxValue {0}", Convert.ToString((int.MaxValue),2));
Console.readLine();
```

Ниже показан результат выполнения этого кода:

```
===== Fun wih Bitwise Operations
6 & 4 = 4 | 100
6 | 4 = 6 | 110
6 ^ 4 = 2 | 10
6 << 1 = 12 | 1100
6 >> 1 = 3 | 11
~6 = -7 | 11111111111111111111111111111001
Int.MaxValue 11111111111111111111111111111111
```

Теперь, когда вам известны основы побитовых операций, самое время применить их к перечислениям. Добавьте в проект новый файл по имени `ContactPreferenceEnum.cs` и приведите его код к такому виду:

```
using System;
namespace FunWithBitwiseOperations
{
  [Flags]
  public enum ContactPreferenceEnum
  {
    None = 1,
    Email = 2,
    Phone = 4,
    Ponyexpress = 6
  }
}
```

Обратите внимание на атрибут `Flags`. Он позволяет объединять множество значений из перечисления в одной переменной. Скажем, вот как можно объединить `Email` и `Phone`:

```
ContactPreferenceEnum emailAndPhone = ContactPreferenceEnum.Email |
ContactPreferenceEnum.Phone;
```

В итоге появляется возможность проверки, присутствует ли одно из значений в объединенном значении. Например, если вы хотите выяснить, имеется ли значение `ContactPreference` в переменной `emailAndPhone`, то можете написать такой код:

```
Console.WriteLine("None? {0}", (emailAndPhone |
ContactPreferenceEnum.None) == emailAndPhone);
Console.WriteLine("Email? {0}", (emailAndPhone |
ContactPreferenceEnum.Email) == emailAndPhone);
Console.WriteLine("Phone? {0}", (emailAndPhone |
ContactPreferenceEnum.Phone) == emailAndPhone);
Console.WriteLine("Text? {0}", (emailAndPhone |
ContactPreferenceEnum.Text) == emailAndPhone);
```

В результате выполнения кода в окне консоли появляется следующий вывод:

```
None? False
Email? True
Phone? True
Text? False
```

Понятие структуры (как типа значения)

Теперь, когда вы понимаете роль типов перечислений, давайте посмотрим, как использовать *структуры* .NET Core. Типы структур хорошо подходят для моделирования в приложении математических, геометрических и других "атомарных" сущностей. Структура (такая как перечисление) — это определяемый пользователем тип; тем не менее, структура не является просто коллекцией пар "имя-значение". Взамен структуры представляют собой типы, которые могут содержать любое количество полей данных и членов, действующих на таких полях.

На заметку! Если вы имеете опыт объектно-ориентированного программирования, тогда можете считать структуры "легковесными типами классов", т.к. они предоставляют способ определения типа, который поддерживает инкапсуляцию, но не может использоваться для построения семейства взаимосвязанных типов. Когда возникает потребность в создании семейства типов, связанных отношением наследования, необходимо применять классы.

На первый взгляд процесс определения и использования структур выглядит простым, но, как часто бывает, самое сложное скрыто в деталях. Чтобы приступить к изучению основ типов структур, создайте новый проект по имени `FunWithStructures`. В языке C# структуры определяются с применением ключевого слова `struct`. Определите новую структуру по имени `Point`, представляющую точку, которая содержит две переменные типа `int` и набор методов для взаимодействия с ними:

```
struct Point
{
  // Поля структуры.
  public int X;
  public int Y;

  // Добавить 1 к позиции (X, Y).
  public void Increment()
  {
    X++; Y++;
  }

  // Вычесть 1 из позиции (X, Y).
  public void Decrement()
  {
    X--; Y--;
  }

  // Отобразить текущую позицию.
  public void Display()
  {
    Console.WriteLine("X = {0}, Y = {1}", X, Y);
  }
}
```

Здесь определены два целочисленных поля (`X` и `Y`) с использованием ключевого слова `public`, которое является модификатором управления доступом (их обсуждение будет продолжено в главе 5). Объявление данных с ключевым словом `public` обеспечивает вызывающему коду возможность прямого доступа к таким данным через переменную типа `Point` (посредством операции точки).

На заметку! Определение открытых данных внутри класса или структуры обычно считается плохим стилем программирования. Взамен рекомендуется определять закрытые данные, доступ и изменение которых производится с применением открытых свойств. Более подробные сведения приведены в главе 5.

Вот код, который позволяет протестировать тип `Point`:

```
Console.WriteLine("***** A First Look at Structures *****\n");

// Создать начальную переменную типа Point.
Point myPoint;
myPoint.X = 349;
myPoint.Y = 76;
myPoint.Display();

// Скорректировать значения X и Y.
myPoint.Increment();
myPoint.Display();
Console.ReadLine();
```

Вывод выглядит вполне ожидаемо:

```
***** A First Look at Structures *****

X = 349, Y = 76
X = 350, Y = 77
```

Создание переменных типа структур

Для создания переменной типа структуры на выбор доступно несколько вариантов. В следующем коде просто создается переменная типа `Point` и затем каждому ее открытому полю данных присваиваются значения до того, как обращаться к членам переменной. Если *не* присвоить значения открытым полям данных (`X` и `Y` в данном случае) перед использованием структуры, то компилятор сообщит об ошибке:

```
// Ошибка! Полю Y не присвоено значение.
Point p1;
p1.X = 10;
p1.Display();

// Все в порядке! Перед использованием значения присвоены обоим полям.
Point p2;
p2.X = 10;
p2.Y = 10;
p2.Display();
```

В качестве альтернативы переменные типа структур можно создавать с применением ключевого слова `new` языка C#, что приводит к вызову *стандартного конструктора* структуры. По определению стандартный конструктор не принимает аргументов. Преимущество вызова стандартного конструктора структуры заключается в том, что каждое поле данных автоматически получает свое стандартное значение:

```
// Установить для всех полей стандартные значения,
// используя стандартный конструктор.
Point p1 = new Point();

// Выводит X=0, Y=0
p1.Display();
```

Допускается также проектировать структуры со *специальным конструктором*, что позволяет указывать значения для полей данных при создании переменной, а не устанавливать их по отдельности. Конструкторы подробно рассматриваются в главе 5; однако в целях иллюстрации измените структуру `Point` следующим образом:

```
struct Point
{
  // Поля структуры.
  public int X;
  public int Y;

  // Специальный конструктор.
  public Point(int xPos, int yPos)
  {
    X = xPos;
    Y = yPos;
  }
  ...
}
```

Затем переменные типа `Point` можно создавать так:

```
// Вызвать специальный конструктор.
Point p2 = new Point(50, 60);

// Выводит X=50,Y=60
p2.Display();
```

Использование структур, допускающих только чтение (нововведение в версии 7.2)

Структуры можно также помечать как допускающие только чтение, если необходимо, чтобы они были *неизменяемыми*. Неизменяемые объекты должны устанавливаться при конструировании и поскольку изменять их нельзя, они могут быть более производительными. В случае объявления структуры как допускающей только чтение все свойства тоже должны быть доступны только для чтения. Но может возникнуть вопрос, как тогда устанавливать свойство, если оно допускает только чтение? Ответ заключается в том, что значения свойств должны устанавливаться во время конструирования структуры. Модифицируйте класс, представляющий точку, как показано ниже:

```
readonly struct ReadOnlyPoint
{
  // Поля структуры.
  public int X { get; }
  public int Y { get; }
  // Отобразить текущую позицию.
  public void Display()
  {
    Console.WriteLine($"X = {X}, Y = {Y}");
  }
  public ReadOnlyPoint(int xPos, int yPos)
  {
    X = xPos;
    Y = yPos;
  }
}
```

Методы `Increment()` и `Decrement()` были удалены, т.к. переменные допускают только чтение. Обратите внимание на свойства `X` и `Y`. Вместо определения их в виде полей они создаются как автоматические свойства, доступные только для чтения. Автоматические свойства рассматриваются в главе 5.

Использование членов, допускающих только чтение (нововведение в версии 8.0)

В версии C# 8.0 появилась возможность объявления индивидуальных полей структуры как `readonly`. Это обеспечивает более высокий уровень детализации, чем объявление целой структуры как допускающей только чтение. Модификатор `readonly` может применяться к методам, свойствам и средствам доступа для свойств. Добавьте следующий код структуры в свой файл за пределами класса `Program`:

```
struct PointWithReadOnly
{
  // Поля структуры.
  public int X;
  public readonly int Y;
  public readonly string Name;

  // Отобразить текущую позицию и название.
  public readonly void Display()
  {
    Console.WriteLine($"X = {X}, Y = {Y}, Name = {Name}");
  }

  // Специальный конструктор.
  public PointWithReadOnly(int xPos, int yPos, string name)
  {
    X = xPos;
    Y = yPos;
    Name = name;
  }
}
```

Для использования этой новой структуры добавьте к операторам верхнего уровня такой код:

```
PointWithReadOnly p3 =
  new PointWithReadOnly(50,60,"Point w/RO");
p3.Display();
```

Использование структур `ref` (нововведение в версии 7.2)

При определении структуры в C# 7.2 также появилась возможность применения модификатора `ref`. Он требует, чтобы все экземпляры структуры находились в стеке и не могли присваиваться свойству другого класса. Формальная причина для этого заключается в том, что ссылки на структуры `ref` из кучи невозможны. Отличие между стеком и кучей объясняется в следующем разделе.

Ниже перечислены дополнительные ограничения структур `ref`:

- их нельзя присваивать переменной типа `object` или `dynamic`, и они не могут быть интерфейсного типа;
- они не могут реализовывать интерфейсы;

- они не могут использоваться в качестве свойства структуры, не являющейся `ref`;
- они не могут применяться в асинхронных методах, итераторах, лямбда-выражениях или локальных функциях.

Показанный далее код, в котором создается простая структура и затем предпринимается попытка создать в этой структуре свойство, типизированное как структура `ref`, не скомпилируется:

```
struct NormalPoint
{
  // Этот код не скомпилируется.
  public PointWithRef PropPointer { get; set; }
}
```

Модификаторы `readonly` и `ref` можно сочетать для получения преимуществ и ограничений их обоих.

Использование освобождаемых структур `ref` (нововведение в версии 8.0)

Как было указано в предыдущем разделе, структуры `ref` (и структуры `ref`, допускающие только чтение) не могут реализовывать интерфейсы, а потому реализовать `IDisposable` нельзя. В версии C# 8.0 появилась возможность делать структуры `ref` и структуры `ref`, допускающие только чтение, освобождаемыми, добавляя открытый метод `void Dispose()`.

Добавьте в главный файл следующее определение структуры:

```
ref struct DisposableRefStruct
{
  public int X;
  public readonly int Y;
  public readonly void Display()
  {
    Console.WriteLine($"X = {X}, Y = {Y}");
  }
  // Специальный конструктор.
  public DisposableRefStruct(int xPos, int yPos)
  {
    X = xPos;
    Y = yPos;
    Console.WriteLine("Created!");       // Экземпляр создан!
  }
  public void Dispose()
  {
    // Выполнить здесь очистку любых ресурсов.
    Console.WriteLine("Disposed!");      // Экземпляр освобожден!
  }
}
```

Теперь поместите в конце операторов верхнего уровня приведенный ниже код, предназначенный для создания и освобождения новой структуры:

```
var s = new DisposableRefStruct(50, 60);
s.Display();
s.Dispose();
```

На заметку! Темы времени жизни и освобождения объектов раскрываются в главе 9.

Чтобы углубить понимание выделения памяти в стеке и куче, необходимо ознакомиться с отличиями между типами значений и ссылочными типами .NET Core.

Типы значений и ссылочные типы

На заметку! В последующем обсуждении типов значений и ссылочных типов предполагается наличие у вас базовых знаний объектно-ориентированного программирования. Если это не так, тогда имеет смысл перейти к чтению раздела "Понятие типов C#, допускающих null" далее в главе и возвратиться к настоящему разделу после изучения глав 5 и 6.

В отличие от массивов, строк и перечислений структуры C# не имеют идентично именованного представления в библиотеке .NET Core (т.е. класс вроде `System.Structure` отсутствует), но они являются неявно производными от абстрактного класса `System.ValueType`. Роль класса `System.ValueType` заключается в обеспечении размещения экземпляра производного типа (например, любой структуры) в *стеке*, а не в *куче* с автоматической сборкой мусора. Выражаясь просто, данные, размещаемые в стеке, могут создаваться и уничтожаться быстро, т.к. время их жизни определяется областью видимости, в которой они объявлены. С другой стороны, данные, размещаемые в куче, отслеживаются сборщиком мусора .NET Core и имеют время жизни, которое определяется многими факторами, объясняемыми в главе 9.

С точки зрения функциональности единственное назначение класса `System.ValueType` — переопределение виртуальных методов, объявленных в классе `System.Object`, с целью использования семантики на основе значений, а не ссылок. Вероятно, вы уже знаете, что переопределение представляет собой процесс изменения реализации виртуального (или возможно абстрактного) метода, определенного внутри базового класса. Базовым классом для `ValueType` является `System.Object`. В действительности методы экземпляра, определенные в `System.ValueType`, идентичны методам экземпляра, которые определены в `System.Object`:

```
// Структуры и перечисления неявно расширяют класс System.ValueType.
public abstract class ValueType : object
{
  public virtual bool Equals(object obj);
  public virtual int GetHashCode();
  public Type GetType();
  public virtual string ToString();
}
```

Учитывая, что типы значений применяют семантику на основе значений, время жизни структуры (что относится ко всем числовым типам данных (`int`, `float`), а также к любому перечислению или структуре) предсказуемо. Когда переменная типа структуры покидает область определения, она немедленно удаляется из памяти:

```
// Локальные структуры извлекаются из стека,
// когда метод возвращает управление.
static void LocalValueTypes()
{
  // Вспомните, что int - на самом деле структура System.Int32.
  int i = 0;
```

```
  // Вспомните, что Point - в действительности тип структуры.
  Point p = new Point();
} // Здесь i и p покидают стек!
```

Использование типов значений, ссылочных типов и операции присваивания

Когда переменная одного типа значения присваивается переменной другого типа значения, выполняется почленное копирование полей данных. В случае простого типа данных, такого как `System.Int32`, единственным копируемым членом будет числовое значение. Однако для типа `Point` в новую переменную структуры будут копироваться значения полей `X` и `Y`. В целях демонстрации создайте новый проект консольного приложения по имени `FunWithValueAndReferenceTypes` и скопируйте предыдущее определение `Point` в новое пространство имен, после чего добавьте к операторам верхнего уровня следующую локальную функцию:

```
// Присваивание двух внутренних типов значений дает
// в результате две независимые переменные в стеке.
static void ValueTypeAssignment()
{
  Console.WriteLine("Assigning value types\n");

  Point p1 = new Point(10, 10);
  Point p2 = p1;

  // Вывести значения обеих переменных Point.
  p1.Display();
  p2.Display();

  // Изменить p1.X и снова вывести значения переменных.
  // Значение p2.X не изменилось.
  p1.X = 100;
  Console.WriteLine("\n=> Changed p1.X\n");
  p1.Display();
  p2.Display();
}
```

Здесь создается переменная типа `Point` (`p1`), которая присваивается другой переменной типа `Point` (`p2`). Поскольку `Point` — тип значения, в стеке находятся две копии `Point`, каждой из которых можно манипулировать независимым образом. Поэтому при изменении значения `p1.X` значение `p2.X` остается незатронутым:

```
Assigning value types

X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 10, Y = 10
```

По контрасту с типами значений, когда операция присваивания применяется к переменным ссылочных типов (т.е. экземплярам всех классов), происходит перенаправление на то, на что ссылочная переменная указывает в памяти. В целях иллюстрации создайте новый класс по имени `PointRef` с теми же членами, что и у структуры `Point`, но только переименуйте конструктор в соответствии с именем данного класса:

```
// Классы всегда являются ссылочными типами.
class PointRef
{
  // Те же самые члены, что и в структуре Point...
  // Не забудьте изменить имя конструктора на PointRef!
  public PointRef(int xPos, int yPos)
  {
    X = xPos;
    Y = yPos;
  }
}
```

Задействуйте готовый тип `PointRef` в следующем новом методе. Обратите внимание, что помимо использования класса `PointRef` вместо структуры `Point` код идентичен коду метода `ValueTypeAssignment()`:

```
static void ReferenceTypeAssignment()
{
  Console.WriteLine("Assigning reference types\n");
  PointRef p1 = new PointRef(10, 10);
  PointRef p2 = p1;

  // Вывести значения обеих переменных PointRef.
  p1.Display();
  p2.Display();

  // Изменить p1.X и снова вывести значения.
  p1.X = 100;
  Console.WriteLine("\n=> Changed p1.X\n");
  p1.Display();
  p2.Display();
}
```

В рассматриваемом случае есть две ссылки, указывающие на тот же самый объект в управляемой куче. Таким образом, когда значение `X` изменяется с использованием ссылки `p1`, изменится также и значение `p2.X`. Вот вывод, получаемый в результате вызова этого нового метода:

```
Assigning reference types

X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 100, Y = 10
```

Использование типов значений, содержащих ссылочные типы

Теперь, когда вы лучше понимаете базовые отличия между типами значений и ссылочными типами, давайте обратимся к более сложному примеру. Предположим, что имеется следующий ссылочный тип (класс), который поддерживает информационную строку (`InfoString`), устанавливаемую с применением специального конструктора:

```
class ShapeInfo
{
  public string InfoString;
```

```
  public ShapeInfo(string info)
  {
    InfoString = info;
  }
}
```

Далее представим, что переменная типа `ShapeInfo` должна содержаться внутри типа значения по имени `Rectangle`. Кроме того, в типе `Rectangle` предусмотрен специальный конструктор, который позволяет вызывающему коду указывать значение для внутренней переменной-члена типа `ShapeInfo`. Вот полное определение типа `Rectangle`:

```
struct Rectangle
{
  // Структура Rectangle содержит член ссылочного типа.
  public ShapeInfo RectInfo;

  public int RectTop, RectLeft, RectBottom, RectRight;

  public Rectangle(string info, int top, int left, int bottom, int right)
  {
    RectInfo = new ShapeInfo(info);
    RectTop = top; RectBottom = bottom;
    RectLeft = left; RectRight = right;
  }

  public void Display()
  {
    Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}, " +
      "Left = {3}, Right = {4}",
      RectInfo.InfoString, RectTop, RectBottom, RectLeft, RectRight);
  }
}
```

Здесь ссылочный тип содержится внутри типа значения. Возникает важный вопрос: что произойдет в результате присваивания одной переменной типа `Rectangle` другой переменной того же типа? Учитывая то, что уже известно о типах значений, можно корректно предположить, что целочисленные данные (которые на самом деле являются структурой — `System.Int32`) должны быть независимой сущностью для каждой переменной `Rectangle`. Но что можно сказать о внутреннем ссылочном типе? Будет ли полностью скопировано *состояние* этого объекта или же только ссылка на него? Чтобы получить ответ, определите следующий метод и вызовите его:

```
static void ValueTypeContainingRefType()
{
  // Создать первую переменную Rectangle.
  Console.WriteLine("-> Creating r1");
  Rectangle r1 = new Rectangle("First Rect", 10, 10, 50, 50);

  // Присвоить новой переменной Rectangle переменную r1.
  Console.WriteLine("-> Assigning r2 to r1");
  Rectangle r2 = r1;

  // Изменить некоторые значения в r2.
  Console.WriteLine("-> Changing values of r2");
  r2.RectInfo.InfoString = "This is new info!";
  r2.RectBottom = 4444;
```

```
  // Вывести значения из обеих переменных Rectangle.
  r1.Display();
  r2.Display();
}
```

Вывод будет таким:

```
-> Creating r1
-> Assigning r2 to r1
-> Changing values of r2
String = This is new info!, Top = 10, Bottom = 50, Left = 10, Right = 50
String = This is new info!, Top = 10, Bottom = 4444, Left = 10, Right = 50
```

Как видите, в случае модификации значения информационной строки с использованием ссылки `r2` для ссылки `r1` отображается то же самое значение. По умолчанию, если тип значения содержит другие ссылочные типы, то присваивание приводит к копированию ссылок. В результате получаются две независимые структуры, каждая из которых содержит ссылку, указывающую на один и тот же объект в памяти (т.е. создается поверхностная копия). Для выполнения глубокого копирования, при котором в новый объект полностью копируется состояние внутренних ссылок, можно реализовать интерфейс `ICloneable` (что будет показано в главе 8).

Передача ссылочных типов по значению

Ранее в главе объяснялось, что ссылочные типы и типы значений могут передаваться методам как параметры. Тем не менее, передача ссылочного типа (например, класса) по ссылке совершенно отличается от его передачи по значению. Чтобы понять разницу, предположим, что есть простой класс `Person`, определенный в новом проекте консольного приложения по имени `FunWithRefTypeValTypeParams`:

```
class Person
{
  public string personName;
  public int personAge;
  // Конструкторы.
  public Person(string name, int age)
  {
    personName = name;
    personAge = age;
  }
  public Person(){}
  public void Display()
  {
    Console.WriteLine("Name: {0}, Age: {1}", personName, personAge);
  }
}
```

А что если мы создадим метод, который позволит вызывающему коду передавать объект `Person` по значению (обратите внимание на отсутствие модификаторов параметров, таких как `out` или `ref`)?

```
static void SendAPersonByValue(Person p)
{
  // Изменить значение возраста в p?
  p.personAge = 99;
```

```
  // Увидит ли вызывающий код это изменение?
  p = new Person("Nikki", 99);
}
```

Здесь видно, что метод `SendAPersonByValue()` пытается присвоить входной ссылке на `Person` новый объект `Person`, а также изменить некоторые данные состояния. Протестируем этот метод с помощью следующего кода:

```
// Передача ссылочных типов по значению.
Console.WriteLine("***** Passing Person object by value *****");
Person fred = new Person("Fred", 12);
Console.WriteLine("\nBefore by value call, Person is:");
                  // Перед вызовом с передачей по значению
fred.Display();

SendAPersonByValue(fred);
Console.WriteLine("\nAfter by value call, Person is:");
                  // После вызова с передачей по значению
fred.Display();
Console.ReadLine();
```

Ниже показан результирующий вывод:

```
***** Passing Person object by value *****

Before by value call, Person is:
Name: Fred, Age: 12

After by value call, Person is:
Name: Fred, Age: 99
```

Легко заметить, что значение `PersoneAge` было изменено. Такое поведение, которое обсуждалось ранее, должно стать более понятным теперь, когда вы знаете, как работают ссылочные типы. Учитывая, что попытка изменения состояния входного объекта `Person` прошла успешно, возникает вопрос: что же тогда было скопировано? Ответ: была получена копия ссылки на объект из вызывающего кода. Следовательно, раз уж метод `SendAPersonByValue()` указывает на тот же самый объект, что и вызывающий код, становится возможным изменение данных состояния этого объекта. Нельзя лишь переустанавливать ссылку так, чтобы она указывала на какой-то *другой* объект.

Передача ссылочных типов по ссылке

Предположим, что имеется метод `SendAPersonByReference()`, в котором ссылочный тип передается по ссылке (обратите внимание на наличие модификатора параметра `ref`):

```
static void SendAPersonByReference(ref Person p)
{
  // Изменить некоторые данные в p.
  p.personAge = 555;

  // p теперь указывает на новый объект в куче!
  p = new Person("Nikki", 999);
}
```

Как и можно было ожидать, вызываемому коду предоставлена полная свобода в плане манипулирования входным параметром. Вызываемый код может не только из-

менять состояние объекта, но и переопределять ссылку так, чтобы она указывала на новый объект Person. Взгляните на следующий обновленный код:

```
// Передача ссылочных типов по ссылке.
Console.WriteLine("***** Passing Person object by reference *****");
...
Person mel = new Person("Mel", 23);
Console.WriteLine("Before by ref call, Person is:");
                // Перед вызовом с передачей по ссылке
mel.Display();
SendAPersonByReference(ref mel);
Console.WriteLine("After by ref call, Person is:");
                // После вызова с передачей по ссылке
mel.Display();
Console.ReadLine();
```

Вот вывод:

```
***** Passing Person object by reference *****
Before by ref call, Person is:
Name: Mel, Age: 23
After by ref call, Person is:
Name: Nikki, Age: 999
```

Здесь видно, что после вызова объект по имени Mel возвращается как объект по имени Nikki, поскольку метод имел возможность изменить то, на что указывала в памяти входная ссылка. Ниже представлены основные правила, которые необходимо соблюдать при передаче ссылочных типов.

- Если ссылочный тип передается по ссылке, тогда вызываемый код может изменять значения данных состояния объекта, а также объект, на который указывает ссылка.
- Если ссылочный тип передается по значению, то вызываемый код может изменять значения данных состояния объекта, но не объект, на который указывает ссылка.

Заключительные детали относительно типов значений и ссылочных типов

В завершение данной темы в табл. 4.4 приведена сводка по основным отличиям между типами значений и ссылочными типами.

Таблица 4.4. Отличия между типами значений и ссылочными типами

Вопрос	Тип значения	Ссылочный тип
Где размещаются объекты?	Размещаются в стеке	Размещаются в управляемой куче
Как представлена переменная?	Переменные типов значений являются локальными копиями	Переменные ссылочных типов указывают на память, занимаемую размещенным экземпляром

Окончание табл. 4.4

Вопрос	Тип значения	Ссылочный тип
Какой тип является базовым?	Неявно расширяет `System.ValueType`	Может быть производным от любого другого типа (кроме `System.ValueType`), если только этот тип не запечатан (см. главу 6)
Может ли этот тип выступать в качестве базового для других типов?	Нет. Типы значений всегда запечатаны, и наследовать от них нельзя	Да. Если тип не запечатан, то он может выступать в качестве базового для других типов
Каково стандартное поведение передачи параметров?	Переменные передаются по значению (т.е. вызываемой функции передается копия переменной)	Для ссылочных типов ссылка копируется по значению
Можно ли переопределить метод `System.Object.Finalize()` в этом типе?	Нет	Да, косвенно (как показано в главе 9)
Можно ли определить конструкторы для этого типа?	Да, но стандартный конструктор является зарезервированным (т.е. все специальные конструкторы должны иметь аргументы)	Конечно!
Когда переменные этого типа прекращают свое существование?	Когда покидают область видимости, в которой они были определены	Когда объект подвергается сборке мусора

Несмотря на различия, типы значений и ссылочные типы могут реализовывать интерфейсы и поддерживать любое количество полей, методов, перегруженных операций, констант, свойств и событий.

Понятие типов C#, допускающих `null`

Давайте исследуем роль *типов данных, допускающих значение* `null`, с применением проекта консольного приложения по имени `FunWithNullableValueTypes`. Как вам уже известно, типы данных C# обладают фиксированным диапазоном значений и представлены в виде типов пространства имен `System`. Например, тип данных `System.Boolean` может принимать только значения из набора (`true`, `false`). Вспомните, что все числовые типы данных (а также `Boolean`) являются *типами значений*. Типам значений никогда не может быть присвоено значение `null`, потому что оно служит для представления пустой объектной ссылки.

```
// Ошибка на этапе компиляции!
// Типы значений нельзя устанавливать в null!
bool myBool = null;
int myInt = null;
```

В языке C# поддерживается концепция *типов данных, допускающих значение* `null`. Выражаясь просто, допускающий `null` тип может представлять все значения лежащего в основе типа плюс `null`. Таким образом, если вы объявите переменную

типа bool, допускающего null, то ей можно будет присваивать значение из набора {true, false, null}. Это может быть чрезвычайно удобно при работе с реляционными базами данных, поскольку в таблицах баз данных довольно часто встречаются столбцы, для которых значения не определены. Без концепции типов данных, допускающих null, в C# не было бы удобного способа для представления числовых элементов данных без значений.

Чтобы определить переменную типа, допускающего null, необходимо добавить к имени интересующего типа данных суффикс в виде знака вопроса (?). До выхода версии C# 8.0 такой синтаксис был законным только в случае применения к типам значений (более подробные сведения ищите в разделе "Использование ссылочных типов, допускающих null" далее в главе). Подобно переменным с типами, не допускающими null, локальным переменным, имеющим типы, которые допускают null, должно присваиваться начальное значение, прежде чем ими можно будет пользоваться:

```
static void LocalNullableVariables()
{
  // Определить несколько локальных переменных
  // с типами, допускающими null.
  int? nullableInt = 10;
  double? nullableDouble = 3.14;
  bool? nullableBool = null;
  char? nullableChar = 'a';
  int?[] arrayOfNullableInts = new int?[10];
}
```

Использование типов значений, допускающих null

В языке C# система обозначений в форме суффикса ? представляет собой сокращение для создания экземпляра обобщенного типа структуры System.Nullable<T>. Она также применяется для создания ссылочных типов, допускающих null, но ее поведение несколько отличается. Хотя подробное исследование обобщений мы отложим до главы 10, сейчас важно понимать, что тип System.Nullable<T> предоставляет набор членов, которые могут применяться всеми типами, допускающими null.

Например, с помощью свойства HasValue или операции != можно программно выяснять, действительно ли переменной, допускающей null, было присвоено значение null. Значение, которое присвоено типу, допускающему null, можно получать напрямую или через свойство Value. Учитывая, что суффикс ? является просто сокращением для использования Nullable<T>, предыдущий метод LocalNullableVariables() можно было бы реализовать следующим образом:

```
static void LocalNullableVariablesUsingNullable()
{
  // Определить несколько типов, допускающих null,
  // с применением Nullable<T>.
  Nullable<int> nullableInt = 10;
  Nullable<double> nullableDouble = 3.14;
  Nullable<bool> nullableBool = null;
  Nullable<char> nullableChar = 'a';
  Nullable<int>[] arrayOfNullableInts = new Nullable<int>[10];
}
```

Как отмечалось ранее, типы данных, допускающие null, особенно полезны при взаимодействии с базами данных, потому что столбцы в таблицах данных могут быть

намеренно оставлены пустыми (скажем, быть неопределенными). В целях демонстрации рассмотрим показанный далее класс, эмулирующий процесс доступа к базе данных с таблицей, в которой два столбца могут принимать значения null. Обратите внимание, что метод GetIntFromDatabase() не присваивает значение члену целочисленного типа, допускающего null, тогда как метод GetBoolFromDatabase() присваивает допустимое значение члену типа bool?.

```
class DatabaseReader
{
  // Поле данных типа, допускающего null.
  public int? numericValue = null;
  public bool? boolValue = true;

  // Обратите внимание на возвращаемый тип, допускающий null.
  public int? GetIntFromDatabase()
  { return numericValue; }

  // Обратите внимание на возвращаемый тип, допускающий null.
  public bool? GetBoolFromDatabase()
  { return boolValue; }
}
```

В следующем коде происходит обращение к каждому члену класса DatabaseReader и выяснение присвоенных значений с применением членов HasValue и Value, а также операции равенства C# (точнее операции "не равно"):

```
Console.WriteLine("***** Fun with Nullable Value Types *****\n");
DatabaseReader dr = new DatabaseReader();

// Получить значение int из "базы данных".
int? i = dr.GetIntFromDatabase();
if (i.HasValue)
{
  Console.WriteLine("Value of 'i' is: {0}", i.Value);
                  // Вывод значения переменной i
}
else
{
  Console.WriteLine("Value of 'i' is undefined.");
                  // Значение переменной i не определено
}
// Получить значение bool из "базы данных".
bool? b = dr.GetBoolFromDatabase();
if (b != null)
{
  Console.WriteLine("Value of 'b' is: {0}", b.Value);
                  // Вывод значения переменной b
}
else
{
  Console.WriteLine("Value of 'b' is undefined.");
                  // Значение переменной b не определено
}
Console.ReadLine();
```

Использование ссылочных типов, допускающих null (нововведение в версии 8.0)

Важным средством, добавленным в версию C# 8, является поддержка ссылочных типов, допускающих значение null. На самом деле изменение было настолько значительным, что инфраструктуру .NET Framework не удалось обновить для поддержки нового средства. В итоге было принято решение поддерживать C# 8 только в .NET Core 3.0 и последующих версиях и также по умолчанию отключить поддержку ссылочных типов, допускающих null. В новом проекте .NET Core 3.0/3.1 или .NET 5 ссылочные типы функционируют точно так же, как в C# 7. Это сделано для того, чтобы предотвратить нарушение работы миллиардов строк кода, существовавших в экосистеме до появления C# 8. Разработчики в своих приложениях должны дать согласие на включение ссылочных типов, допускающих null.

Ссылочные типы, допускающие null, подчиняются множеству тех же самых правил, что и типы значений, допускающие null. Переменным ссылочных типов, не допускающих null, во время инициализации должны присваиваться отличающиеся от null значения, которые позже нельзя изменять на null. Переменные ссылочных типов, допускающих null, могут принимать значение null, но перед первым использованием им по-прежнему должны присваиваться какие-то значения (либо фактический экземпляр чего-нибудь, либо значение null).

Для указания способности иметь значение null в ссылочных типах, допускающих null, применяется тот же самый символ ?. Однако он не является сокращением для использования System.Nullable<T>, т.к. на месте T могут находиться только типы значений. Не забывайте, что обобщения и ограничения рассматриваются в главе 10.

Включение ссылочных типов, допускающих null

Поддержка для ссылочных типов, допускающих null, управляется установкой контекста допустимости значения null. Это может распространяться на целый проект (за счет обновления файла проекта) или охватывать лишь несколько строк (путем применения директив компилятора). Вдобавок можно устанавливать следующие два контекста.

- Контекст с заметками о допустимости значения null: включает/отключает заметки о допустимости null (?) для ссылочных типов, допускающих null.
- Контекст с предупреждениями о допустимости значения null: включает/отключает предупреждения компилятора для ссылочных типов, допускающих null.

Чтобы увидеть их в действии, создайте новый проект консольного приложения по имени FunWithNullableReferenceTypes. Откройте файл проекта (если вы используете Visual Studio, тогда дважды щелкните на имени проекта в окне Solution Explorer или щелкните правой кнопкой мыши на имени проекта и выберите в контекстном меню пункт Edit Project file (Редактировать файл проекта)). Модифицируйте содержимое файла проекта для поддержки ссылочных типов, допускающих null, за счет добавления элемента <Nullable> (все доступные варианты представлены в табл. 4.5).

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Таблица 4.5. Значения для элемента <Nullable> в файлах проектов

Значение	Описание
enable	Заметки о допустимости значения null включены; предупреждения о допустимости значения null включены
warnings	Заметки о допустимости значения null отключены; предупреждения о допустимости значения null включены
annotations	Заметки о допустимости значения null включены; предупреждения о допустимости значения null отключены
disable	Заметки о допустимости значения null отключены; предупреждения о допустимости значения null отключены

Элемент <Nullable> оказывает влияние на весь проект. Для управления меньшими частями проекта используйте директиву компилятора #nullable, значения которой описаны в табл. 4.6.

Таблица 4.6. Значения для директивы компилятора #nullable

Значение	Описание
enable	Заметки включены; предупреждения включены
disable	Заметки отключены; предупреждения отключены
restore	Возврат всех настроек к настройкам из файла проекта
disable warnings	Предупреждения отключены; заметки не затронуты
enable warnings	Предупреждения включены; заметки не затронуты
restore warnings	Предупреждения сброшены к настройкам из файла проекта; заметки не затронуты
disable annotations	Заметки отключены; предупреждения не затронуты
enable annotations	Заметки включены; предупреждения не затронуты
restore annotations	Заметки сброшены к настройкам из файла проекта; предупреждения не затронуты

Ссылочные типы, допускающие null, в действии

Во многом из-за важности изменения ошибки с типами, допускающими значение null, возникают только при их ненадлежащем применении. Добавьте в файл Program.cs следующий класс:

```
public class TestClass
{
  public string Name { get; set; }
  public int Age { get; set; }
}
```

Как видите, это просто нормальный класс. Возможность принятия значения null появляется при использовании данного класса в коде. Взгляните на показанные ниже объявления:

```
string? nullableString = null;
TestClass? myNullableClass = null;
```

Настройка в файле проекта помещает весь проект в контекст допустимости значения null, который разрешает применение объявлений типов string и TestClass с заметками о допустимости значения null (?). Следующая строка кода вызывает генерацию предупреждения (CS8600) из-за присваивания null типу, не допускающему значение null, в контексте допустимости значения null:

```
// Предупреждение CS8600 Converting null literal or possible null
// value to non-nullable type
//     Преобразование литерала null или возможного значения null
//     в тип, не допускающий null
TestClass myNonNullableClass = myNullableClass;
```

Для более точного управления тем, где в проекте находятся контексты допустимости значения null, с помощью директивы компилятора #nullable можно включать или отключать контекст (как обсуждалось ранее). В приведенном далее коде контекст допустимости значения null (установленный на уровне проекта) сначала отключается, после чего снова включается за счет восстановления настройки из файла проекта:

```
#nullable disable
TestClass anotherNullableClass = null;
// Предупреждение CS8632 The annotation for nullable reference types
// should only be used in code within a '#nullable' annotations
//       Заметка для ссылочных типов, допускающих значение null,
//       должна использоваться только в коде внутри
//       #nullable enable annotations
TestClass? badDefinition = null;
// Предупреждение CS8632 The annotation for nullable reference types
// should only be used in code within a '#nullable' annotations
//       Заметка для ссылочных типов, допускающих значение null,
//       должна использоваться только в коде внутри
//       #nullable enable annotations
string? anotherNullableString = null;
#nullable restore
```

В заключение важно отметить, что ссылочные типы, допускающие значение null, не имеют свойств HasValue и Value, т.к. они предоставляются System.Nullable<T>.

Рекомендации по переносу кода

Если при переносе кода из C# 7 в C# 8 или C# 9 вы хотите задействовать ссылочные типы, допускающие значение null, то можете использовать для работы с кодом комбинацию настройки проекта и директив компилятора. Общепринятая практика предусматривает первоначальное включение предупреждений и отключение заметок о допустимости значения null для всего проекта. Затем по мере приведения в порядок областей кода применяйте директивы компилятора для постепенного включения заметок.

Работа с типами, допускающими значение null

Для работы с типами, допускающими значение null, в языке C# предлагается несколько операций. В последующих разделах рассматриваются операция объединения с null, операция присваивания с объединением с null и null-условная операция. Для проработки примеров используйте ранее созданный проект FunWithNullableValueTypes.

Операция объединения с null

Следующий важный аспект связан с тем, что любая переменная, которая может иметь значение null (т.е. переменная ссылочного типа или переменная типа, допускающего null), может использоваться с операцией ?? языка C#, формально называемой *операцией объединения с null*. Операция ?? позволяет присваивать значение типу, допускающему null, если извлеченное значение на самом деле равно null. В рассматриваемом примере мы предположим, что в случае возвращения методом GetIntFromDatabase() значения null (конечно, данный метод запрограммирован так, что он *всегда* возвращает null, но общую идею вы должны уловить) локальной переменной целочисленного типа, допускающего null, необходимо присвоить значение 100. Возвратитесь к проекту NullableValueTypes (сделайте его стартовым) и введите следующий код:

```
// Для краткости код не показан.
Console.WriteLine("***** Fun with Nullable Data *****\n");
DatabaseReader dr = new DatabaseReader();

// Если значение, возвращаемое из GetIntFromDatabase(), равно
// null, тогда присвоить локальной переменной значение 100.
int myData = dr.GetIntFromDatabase() ?? 100;
Console.WriteLine("Value of myData: {0}", myData); //Вывод значения myData
Console.ReadLine();
```

Преимущество применения операции ?? заключается в том, что она дает более компактную версию кода, чем традиционный условный оператор if/else. Однако при желании можно было бы написать показанный ниже функционально эквивалентный код, который в случае возвращения null обеспечит установку переменной в значение 100:

```
// Более длинный код, в котором не используется синтаксис ??.
int? moreData = dr.GetIntFromDatabase();
if (!moreData.HasValue)
{
  moreData = 100;
}
Console.WriteLine("Value of moreData: {0}", moreData);
                // Вывод значения moreData
```

Операция присваивания с объединением с null (нововведение в версии 8.0)

В версии C# 8 появилась *операция присваивания с объединением с null* (??=), основанная на операции объединения с null. Эта операция выполняет присваивание левого операнда правому операнду, только если левый операнд равен null.В качестве примера введите такой код:

```
// Операция присваивания с объединением с null
int? nullableInt = null;
nullableInt ??= 12;
nullableInt ??= 14;
Console.WriteLine(nullableInt);
```

Сначала переменная nullableInt инициализируется значением null. В следующей строке переменной nullableInt присваивается значение 12, поскольку ле-

вый операнд действительно равен null. Но в следующей за ней строке переменной nullableInt *не* присваивается значение 14, т.к. она не равна null.

null-условная операция

При разработке программного обеспечения обычно производится проверка на предмет null входных параметров, которым передаются значения, возвращаемые членами типов (методами, свойствами, индексаторами). Например, пусть имеется метод, который принимает в качестве единственного параметра строковый массив. В целях безопасности его желательно проверять на предмет null, прежде чем начинать обработку. Поступая подобным образом, мы не получим ошибку во время выполнения, если массив окажется пустым. Следующий код демонстрирует традиционный способ реализации такой проверки:

```
static void TesterMethod(string[] args)
{
  // Перед доступом к данным массива мы должны проверить его
  // на равенство null!
  if (args != null)
  {
    Console.WriteLine($"You sent me {args.Length} arguments.");
                      // Вывод количества аргументов
  }
}
```

Чтобы устранить обращение к свойству Length массива string в случае, когда он равен null, здесь используется условный оператор. Если вызывающий код не создаст массив данных и вызовет метод TesterMethod() примерно так, как показано ниже, то никаких ошибок во время выполнения не возникнет:

```
TesterMethod(null);
```

В языке C# имеется маркер null-условной операции (знак вопроса, находящийся после типа переменной, но перед операцией доступа к члену), который позволяет упростить представленную ранее проверку на предмет null. Вместо явного условного оператора, проверяющего на неравенство значению null, теперь можно написать такой код:

```
static void TesterMethod(string[] args)
{
  // Мы должны проверять на предмет null перед доступом к данным массива!
  Console.WriteLine($"You sent me {args?.Length} arguments.");
}
```

В этом случае условный оператор не применяется. Взамен к переменной массива string в качестве суффикса добавлена операция ?. Если переменная args равна null, тогда обращение к свойству Length не приведет к ошибке во время выполнения. Чтобы вывести действительное значение, можно было бы воспользоваться операцией объединения с null и установить стандартное значение:

```
Console.WriteLine($"You sent me {args?.Length ?? 0} arguments.");
```

Существуют дополнительные области написания кода, в которых null-условная операция окажется очень удобной, особенно при работе с делегатами и событиями. Данные темы раскрываются позже в книге (см. главу 12) и вы встретите еще много примеров.

Понятие кортежей (нововведение и обновление в версии 7.0)

В завершение главы мы исследуем роль кортежей, используя проект консольного приложения по имени `FunWithTuples`. Как упоминалось ранее в главе, одна из целей применения параметров `out` — получение более одного значения из вызова метода. Еще один способ предусматривает использование конструкции под названием *кортежи*.

Кортежи, которые являются легковесными структурами данных, содержащими множество полей, фактически появились в версии C# 6, но применяться могли в крайне ограниченной манере. Кроме того, в их реализации C# 6 существовала значительная проблема: каждое поле было реализовано как ссылочный тип, что потенциально порождало проблемы с памятью и/или производительностью (из-за упаковки/распаковки).

В версии C# 7 кортежи вместо ссылочных типов используют новый тип данных `ValueTuple`, сберегая значительных объем памяти. Тип данных `ValueTuple` создает разные структуры на основе количества свойств для кортежа. Кроме того, в C# 7 каждому свойству кортежа можно назначать специфическое имя (подобно переменным), что значительно повышает удобство работы с ними.

Относительно кортежей важно отметить два момента:

- поля не подвергаются проверке достоверности;
- определять собственные методы нельзя.

В действительности кортежи предназначены для того, чтобы служить легковесным механизмом передачи данных.

Начало работы с кортежами

Итак, достаточно теории, давайте напишем какой-нибудь код! Чтобы создать кортеж, просто повестите значения, подлежащие присваиванию, в круглые скобки:

```
("a", 5, "c")
```

Обратите внимание, что все значения не обязаны относиться к тому же самому типу данных. Конструкция с круглыми скобками также применяется для присваивания кортежа переменной (или можно использовать ключевое слово `var` и тогда компилятор назначит типы данных самостоятельно). Показанные далее две строки кода делают одно и то же — присваивают предыдущий пример кортежа переменной. Переменная `values` будет кортежем с двумя свойствами `string` и одним свойством `int`.

```
(string, int, string) values = ("a", 5, "c");
var values = ("a", 5, "c");
```

По умолчанию компилятор назначает каждому свойству имя `ItemX`, где `X` представляет позицию свойства в кортеже, начиная с 1. В предыдущем примере свойства именуются как `Item1`, `Item2` и `Item3`. Доступ к ним осуществляется следующим образом:

```
Console.WriteLine($"First item: {values.Item1}");   // Первый элемент
Console.WriteLine($"Second item: {values.Item2}");  // Второй элемент
Console.WriteLine($"Third item: {values.Item3}");   // Третий элемент
```

Кроме того, к каждому свойству кортежа справа или слева можно добавить специфическое имя. Хотя назначение имен в обеих частях оператора не приводит к ошибке на этапе компиляции, имена в правой части игнорируются, а использоваться будут имена в левой части. Показанные ниже две строки кода демонстрируют установку имен в левой и правой частях оператора, давая тот же самый результат:

```
(string FirstLetter, int TheNumber, string SecondLetter)
  valuesWithNames = ("a", 5, "c");
var valuesWithNames2 = (FirstLetter: "a", TheNumber: 5, SecondLetter: "c");
```

Теперь доступ к свойствам кортежа возможен с применением имен полей, а также системы обозначений `ItemX`:

```
Console.WriteLine($"First item: {valuesWithNames.FirstLetter}");
Console.WriteLine($"Second item: {valuesWithNames.TheNumber}");
Console.WriteLine($"Third item: {valuesWithNames.SecondLetter}");

// Система обозначений ItemX по-прежнему работает!
Console.WriteLine($"First item: {valuesWithNames.Item1}");
Console.WriteLine($"Second item: {valuesWithNames.Item2}");
Console.WriteLine($"Third item: {valuesWithNames.Item3}");
```

Обратите внимание, что при назначении имен в правой части оператора должно использоваться ключевое слово `var` для объявления переменной. Установка типов данных специальным образом (даже без специфических имен) заставляет компилятор применять синтаксис в левой части оператора, назначать свойствам имена согласно системе обозначений `ItemX` и игнорировать имена, указанные в правой части. В следующих двух операторах имена `Custom1` и `Custom2` игнорируются:

```
(int, int) example = (Custom1:5, Custom2:7);
(int Field1, int Field2) example = (Custom1:5, Custom2:7);
```

Важно также понимать, что специальные имена полей существуют только на этапе компиляции и не доступны при инспектировании кортежа во время выполнения с использованием рефлексии (рефлексия раскрывается в главе 17).

Кортежи также могут быть вложенными как кортежи внутри кортежей. Поскольку с каждым свойством в кортеже связан тип данных, и кортеж является типом данных, следующий код полностью законен:

```
Console.WriteLine("=> Nested Tuples");
var nt = (5, 4, ("a", "b"));
```

Использование выведенных имен переменных (обновление в версии C# 7.1)

В C# 7.1 появилась возможность выводить имена переменных кортежей, как показано ниже:

```
Console.WriteLine("=> Inferred Tuple Names");
var foo = new {Prop1 = "first", Prop2 = "second"};
var bar = (foo.Prop1, foo.Prop2);
Console.WriteLine($"{bar.Prop1};{bar.Prop2}");
```

Понятие эквивалентности/неэквивалентности кортежей (нововведение в версии 7.3)

Дополнительным средством в версии C# 7.1 является эквивалентность (==) и неэквивалентность (!=) кортежей. При проверке на неэквивалентность операции сравнения будут выполнять неявные преобразования типов данных внутри кортежей, включая сравнение допускающих и не допускающих `null` кортежей и/или свойств. Это означает, что следующие проверки нормально работают, несмотря на разницу между `int` и `long`:

```
Console.WriteLine("=> Tuples Equality/Inequality");
// Поднятые преобразования.
var left = (a: 5, b: 10);
(int? a, int? b) nullableMembers = (5, 10);
Console.WriteLine(left == nullableMembers);      // Тоже True
// Преобразованным типом слева является (long, long).
(long a, long b) longTuple = (5, 10);
Console.WriteLine(left == longTuple);            // Тоже True
// Преобразования выполняются с кортежами (long, long).
(long a, int b) longFirst = (5, 10);
(int a, long b) longSecond = (5, 10);
Console.WriteLine(longFirst == longSecond);      // Тоже True
```

Кортежи, которые содержат кортежи, также можно сравнивать, но только если они имеют одну и ту же форму. Нельзя сравнивать кортеж с тремя свойствами `int` и кортеж, содержащий два свойства `int` плюс кортеж.

Использование кортежей как возвращаемых значений методов

Ранее в главе для возвращения из вызова метода более одного значения применялись параметры `out`. Для этого существуют другие способы вроде создания класса или структуры специально для возвращения значений. Но если такой класс или структура используется только в целях передачи данных для одного метода, тогда нет нужды выполнять излишнюю работу и писать добавочный код. Кортежи прекрасно подходят для решения задачи, т.к. они легковесны, просты в объявлении и несложны в применении.

Ниже представлен один из примеров, рассмотренных в разделе о параметрах `out`. Метод `FillTheseValues()` возвращает три значения, но требует использования в вызывающем коде трех параметров как механизма передачи:

```
static void FillTheseValues(out int a, out string b, out bool c)
{
  a = 9;
  b = "Enjoy your string.";
  c = true;
}
```

За счет применения кортежа от параметров можно избавиться и все равно получать обратно три значения:

```
static (int a,string b,bool c) FillTheseValues()
{
  return (9,"Enjoy your string.",true);
}
```

Вызывать новый метод не сложнее любого другого метода:

```
var samples = FillTheseValues();
Console.WriteLine($"Int is: {samples.a}");
Console.WriteLine($"String is: {samples.b}");
Console.WriteLine($"Boolean is: {samples.c}");
```

Возможно, даже лучшим примером будет разбор полного имени на отдельные части (имя (`first`), отчество (`middle`), фамилия (`last`)). Следующий метод `SplitNames()` получает полное имя и возвращает кортеж с составными частями:

```
static (string first, string middle, string last) SplitNames(string fullName)
{
  // Действия, необходимые для расщепления полного имени.
  return ("Philip", "F", "Japikse");
}
```

Использование отбрасывания с кортежами

Продолжим пример с методом `SplitNames()`. Пусть известно, что требуются только имя и фамилия, но не отчество. В таком случае можно указать имена свойств для значений, которые необходимо возвращать, а ненужные значения заменить заполнителем в виде подчеркивания (_):

```
var (first, _, last) = SplitNames("Philip F Japikse");
Console.WriteLine($"{first}:{last}");
```

Значение, соответствующее отчеству, в кортеже отбрасывается.

Использование выражений `switch` с сопоставлением с образцом для кортежей (нововведение в версии 8.0)

Теперь, когда вы хорошо разбираетесь в кортежах, самое время возвратиться к примеру выражения `switch` с кортежами, который приводился в конце главы 3:

```
// Выражения switch с кортежами.
static string RockPaperScissors(string first, string second)
{
  return (first, second) switch
  {
    ("rock", "paper") => "Paper wins.",
    ("rock", "scissors") => "Rock wins.",
    ("paper", "rock") => "Paper wins.",
    ("paper", "scissors") => "Scissors wins.",
    ("scissors", "rock") => "Rock wins.",
    ("scissors", "paper") => "Scissors wins.",
    (_, _) => "Tie.",
  };
}
```

В этом примере два параметра преобразуются в кортеж, когда передаются выражению `switch`. В выражении `switch` представлены подходящие значения, а все остальные случаи обрабатывает последний кортеж, состоящий из двух символов отбрасывания.

Сигнатуру метода `RockPaperScissors()` можно было бы записать так, чтобы метод принимал кортеж, например:

```
static string RockPaperScissors(
  (string first, string second) value)
{
  return value switch
  {
    // Для краткости код не показан.
  };
}
```

Деконструирование кортежей

Деконструирование является термином, описывающим отделение свойств кортежа друг от друга с целью применения по одному. Именно это делает метод `FillTheseValues()`. Но есть и другой случай использования такого приема — деконструирование специальных типов.

Возьмем укороченную версию структуры `Point`, которая применялась ранее в главе. В нее был добавлен новый метод по имени `Deconstruct()`, возвращающий индивидуальные свойства экземпляра `Point` в виде кортежа со свойствами `XPos` и `YPos`:

```
struct Point
{
  // Поля структуры.
  public int X;
  public int Y;
  // Специальный конструктор.
  public Point(int XPos, int YPos)
  {
    X = XPos;
    Y = YPos;
  }
  public (int XPos, int YPos) Deconstruct() => (X, Y);
}
```

Новый метод `Deconstruct()` выделен полужирным. Его можно именовать как угодно, но обычно он имеет имя `Deconstruct()`. В результате с помощью единственного вызова метода можно получить индивидуальные значения структуры путем возвращения кортежа:

```
Point p = new Point(7,5);
var pointValues = p.Deconstruct();
Console.WriteLine($"X is: {pointValues.XPos}");
Console.WriteLine($"Y is: {pointValues.YPos}");
```

Деконструирование кортежей с позиционным сопоставлением с образцом (нововведение в версии 8.0)

Когда кортежи имеют доступный метод `Deconstruct()`, деконструирование можно применять в выражении `switch`, основанном на кортежах. Следующий код полагается на пример `Point` и использует значения сгенерированного кортежа в конструкциях `when` выражения `switch`:

```
static string GetQuadrant1(Point p)
{
  return p.Deconstruct() switch
  {
    (0, 0) => "Origin",
```

```
    var (x, y) when x > 0 && y > 0 => "One",
    var (x, y) when x < 0 && y > 0 => "Two",
    var (x, y) when x < 0 && y < 0 => "Three",
    var (x, y) when x > 0 && y < 0 => "Four",
    var (_, _) => "Border",
  };
}
```

Если метод `Deconstruct()` определен с двумя параметрами `out`, тогда выражение `switch` будет автоматически деконструировать экземпляр `Point`. Добавьте к `Point` еще один метод `Deconstruct()`:

```
public void Deconstruct(out int XPos, out int YPos)
  => (XPos,YPos)=(X, Y);
```

Теперь можно модифицировать (или добавить новый) метод `GetQuadrant()`, как показано ниже:

```
static string GetQuadrant2(Point p)
{
  return p switch
  {
    (0, 0) => "Origin",
    var (x, y) when x > 0 && y > 0 => "One",
    var (x, y) when x < 0 && y > 0 => "Two",
    var (x, y) when x < 0 && y < 0 => "Three",
    var (x, y) when x > 0 && y < 0 => "Four",
    var (_, _) => "Border",
  };
}
```

Изменение очень тонкое (и выделено полужирным). В выражении `switch` вместо вызова `p.Deconstruct()` применяется просто переменная `Point`.

Резюме

Глава начиналась с исследования массивов. Затем обсуждались ключевые слова C#, которые позволяют строить специальные методы. Вспомните, что по умолчанию параметры передаются по значению; тем не менее, параметры можно передавать и по ссылке, пометив их модификаторами `ref` или `out`. Кроме того, вы узнали о роли необязательных и именованных параметров, а также о том, как определять и вызывать методы, принимающие массивы параметров.

После рассмотрения темы перегрузки методов в главе приводились подробные сведения, касающиеся способов определения перечислений и структур в C# и их представления в библиотеках базовых классов .NET Core. Попутно рассматривались основные характеристики типов значений и ссылочных типов, включая их поведение при передаче в качестве параметров методам, а также способы взаимодействия с типами данных, допускающими `null`, и переменными, которые могут иметь значение `null` (например, переменными ссылочных типов и переменными типов значений, допускающих `null`), с использованием операций `?`, `??` и `??=`.

Финальный раздел был посвящен давно ожидаемому средству в языке C# — кортежам. После выяснения, что они собой представляют и как работают, кортежи применялись для возвращения множества значений из методов и для деконструирования специальных типов. В главе 5 вы начнете погружаться в детали объектно-ориентированного программирования.

ЧАСТЬ III

Объектно-ориентированное программирование на C#

ГЛАВА 5

Инкапсуляция

В главах 3 и 4 было исследовано несколько основных синтаксических конструкций, присущих любому приложению .NET Core, которое вам придется разрабатывать. Начиная с данной главы, мы приступаем к изучению объектно-ориентированных возможностей языка C#. Первым, что вам предстоит узнать, будет процесс построения четко определенных типов классов, которые поддерживают любое количество *конструкторов*. После введения в основы определения классов и размещения объектов остаток главы будет посвящен теме *инкапсуляции*. В ходе изложения вы научитесь определять свойства классов, а также ознакомитесь с подробными сведениями о ключевом слове `static`, синтаксисе инициализации объектов, полях только для чтения, константных данных и частичных классах.

Знакомство с типом класса C#

С точки зрения платформы .NET Core наиболее фундаментальной программной конструкцией является *тип класса*. Формально класс — это определяемый пользователем тип, состоящий из полей данных (часто называемых *переменными-членами*) и членов, которые оперируют полями данных (к ним относятся конструкторы, свойства, методы, события и т.д.). Коллективно набор полей данных представляет "состояние" экземпляра класса (также известного как *объект*). Мощь объектно-ориентированных языков, таких как C#, заключается в том, что за счет группирования данных и связанной с ними функциональности в унифицированное определение класса вы получаете возможность моделировать свое программное обеспечение в соответствии с сущностями реального мира.

Для начала создайте новый проект консольного приложения C# по имени `SimpleClassExample`. Затем добавьте в проект новый файл класса (`Car.cs`). Поместите в файл `Car.cs` оператор `using` и определите пространство имен, как показано ниже:

```
using System;
namespace SimpleClassExample
{
}
```

На заметку! В приводимых далее примерах определять пространство имен строго обязательно. Однако рекомендуется выработать привычку использовать пространства имен во всем коде, который вы будете писать. Пространства имен подробно обсуждались в главе 1.

Класс определяется в C# с применением ключевого слова `class`. Вот как выглядит простейшее объявление класса (позаботьтесь о том, чтобы объявление класса находилось внутри пространства имен `SimpleClassExample`):

```
class Car
{
}
```

После определения типа класса необходимо определить набор переменных-членов, которые будут использоваться для представления его состояния. Например, вы можете принять решение, что объекты `Car` (автомобили) должны иметь поле данных типа `int`, представляющее текущую скорость, и поле данных типа `string` для представления дружественного названия автомобиля. С учетом таких начальных проектных положений класс `Car` будет выглядеть следующим образом:

```
class Car
{
  // 'Состояние' объекта Car.
  public string petName;
  public int currSpeed;
}
```

Обратите внимание, что переменные-члены объявлены с применением модификатора доступа `public`. Открытые (`public`) члены класса доступны напрямую после того, как создан *объект* этого типа. Вспомните, что термин *объект* используется для описания экземпляра заданного типа класса, который создан с помощью ключевого слова `new`.

На заметку! Поля данных класса редко (если вообще когда-нибудь) должны определяться как открытые. Чтобы обеспечить целостность данных состояния, намного лучше объявлять данные закрытыми (`private`) или возможно защищенными (`protected`) и разрешать контролируемый доступ к данным через свойства (как будет показано далее в главе). Тем не менее, для максимального упрощения первого примера мы определили поля данных как открытые.

После определения набора переменных-членов, представляющих состояние класса, следующим шагом в проектировании будет установка членов, которые моделируют его поведение. Для этого примера в классе `Car` определены методы по имени `SpeedUp()` и `PrintState()`. Модифицируйте код класса `Car` следующим образом:

```
class Car
{
  // 'Состояние' объекта Car.
  public string petName;
  public int currSpeed;

  // Функциональность Car.
  // Использовать синтаксис членов, сжатых до выражений,
  // который рассматривался в главе 4.
  public void PrintState()
    => Console.WriteLine("{0} is going {1} MPH.", petName, currSpeed);

  public void SpeedUp(int delta)
    => currSpeed += delta;
}
```

Метод `PrintState()` — простая диагностическая функция, которая выводит текущее состояние объекта `Car` в окно командной строки. Метод `SpeedUp()` увеличивает скорость автомобиля, представляемого объектом `Car`, на величину, которая

передается во входном параметре типа int. Обновите операторы верхнего уровня в файле Program.cs, как показано ниже:

```
Console.WriteLine("***** Fun with Class Types *****\n");
// Разместить в памяти и сконфигурировать объект Car.
Car myCar = new Car();
myCar.petName = "Henry";
myCar.currSpeed = 10;
// Увеличить скорость автомобиля в несколько раз и вывести новое состояние.
for (int i = 0; i <= 10; i++)
{
  myCar.SpeedUp(5);
  myCar.PrintState();
}
Console.ReadLine();
```

Запустив программу, вы увидите, что переменная Car (myCar) поддерживает свое текущее состояние на протяжении жизни приложения:

```
***** Fun with Class Types *****
Henry is going 15 MPH.
Henry is going 20 MPH.
Henry is going 25 MPH.
Henry is going 30 MPH.
Henry is going 35 MPH.
Henry is going 40 MPH.
Henry is going 45 MPH.
Henry is going 50 MPH.
Henry is going 55 MPH.
Henry is going 60 MPH.
Henry is going 65 MPH.
```

Размещение объектов с помощью ключевого слова new

Как было показано в предыдущем примере кода, объекты должны размещаться в памяти с применением ключевого слова new. Если вы не укажете ключевое слово new и попытаетесь использовать переменную класса в последующем операторе кода, то получите ошибку на этапе компиляции. Например, приведенные далее операторы верхнего уровня не скомпилируются:

```
Console.WriteLine("***** Fun with Class Types *****\n");
// Ошибка на этапе компиляции! Забыли использовать new для создания объекта!
Car myCar;
myCar.petName = "Fred";
```

Чтобы корректно создать объект с применением ключевого слова new, можно определить и разместить в памяти объект Car в одной строке кода:

```
Console.WriteLine("***** Fun with Class Types *****\n");
Car myCar = new Car();
myCar.petName = "Fred";
```

В качестве альтернативы определение и размещение в памяти экземпляра класса может осуществляться в отдельных строках кода:

```
Console.WriteLine("***** Fun with Class Types *****\n");
Car myCar;
```

```
myCar = new Car();
myCar.petName = "Fred";
```

Здесь первый оператор кода просто объявляет *ссылку* на определяемый объект типа `Car`. Ссылка будет указывать на действительный объект в памяти только после ее явного присваивания.

В любом случае к настоящему моменту мы имеем простейший класс, в котором определено несколько элементов данных и ряд базовых операций. Чтобы расширить функциональность текущего класса `Car`, необходимо разобраться с ролью *конструкторов*.

Понятие конструкторов

Учитывая наличие у объекта состояния (представленного значениями его переменных-членов), обычно желательно присвоить подходящие значения полям объекта перед тем, как работать с ним. В настоящее время класс `Car` требует присваивания значений полям `petName` и `currSpeed` по отдельности. Для текущего примера такое действие не слишком проблематично, поскольку открытых элементов данных всего два. Тем не менее, зачастую класс содержит несколько десятков полей, с которыми надо что-то делать. Ясно, что было бы нежелательно писать 20 операторов инициализации для всех 20 элементов данных.

К счастью, язык C# поддерживает использование *конструкторов*, которые позволяют устанавливать состояние объекта в момент его создания. Конструктор — это специальный метод класса, который неявно вызывается при создании объекта с применением ключевого слова `new`. Однако в отличие от "нормального" метода конструктор никогда не имеет возвращаемого значения (даже `void`) и всегда именуется идентично имени класса, объекты которого он конструирует.

Роль стандартного конструктора

Каждый класс C# снабжается "бесплатным" *стандартным конструктором*, который в случае необходимости может быть переопределен. По определению стандартный конструктор никогда не принимает аргументов. После размещения нового объекта в памяти стандартный конструктор гарантирует установку всех полей данных в соответствующие стандартные значения (стандартные значения для типов данных C# были описаны в главе 3).

Если вас не устраивают такие стандартные присваивания, тогда можете переопределить стандартный конструктор в соответствии со своими нуждами. В целях иллюстрации модифицируем класс C# следующим образом:

```
class Car
{
  // 'Состояние' объекта Car.
  public string petName;
  public int currSpeed;
  // Специальный стандартный конструктор.
  public Car()
  {
    petName = "Chuck";
    currSpeed = 10;
  }
  ...
}
```

В данном случае мы заставляем объекты Car начинать свое существование под именем Chuck и со скоростью 10 миль в час. Создать объект Car со стандартными значениями можно так:

```
Console.WriteLine("***** Fun with Class Types *****\n");
// Вызов стандартного конструктора.
Car chuck = new Car();
// Выводит строку "Chuck is going 10 MPH."
chuck.PrintState();
...
```

Определение специальных конструкторов

Обычно помимо стандартного конструктора в классах определяются дополнительные конструкторы. Тем самым пользователю объекта предоставляется простой и согласованный способ инициализации состояния объекта прямо во время его создания. Взгляните на следующее изменение класса Car, который теперь поддерживает в совокупности три конструктора:

```
class Car
{
  // 'Состояние' объекта Car.
  public string petName;
  public int currSpeed;

  // Специальный стандартный конструктор.
  public Car()
  {
    petName = "Chuck";
    currSpeed = 10;
  }

  // Здесь currSpeed получает стандартное значение для типа int (0).
  public Car(string pn)
  {
    petName = pn;
  }

  // Позволяет вызывающему коду установить полное состояние объекта Car.
  public Car(string pn, int cs)
  {
    petName = pn;
    currSpeed = cs;
  }
  ...
}
```

Имейте в виду, что один конструктор отличается от другого (с точки зрения компилятора C#) числом и/или типами аргументов. Вспомните из главы 4, что определение метода с тем же самым именем, но разным количеством или типами аргументов, называется *перегрузкой* метода. Таким образом, конструктор класса Car перегружен, чтобы предложить несколько способов создания объекта во время объявления. В любом случае теперь есть возможность создавать объекты Car, используя любой из его открытых конструкторов. Вот пример:

```
Console.WriteLine("***** Fun with Class Types *****\n");

// Создать объект Car по имени Chuck со скоростью 10 миль в час.
Car chuck = new Car();
chuck.PrintState();

// Создать объект Car по имени Mary со скоростью 0 миль в час.
Car mary = new Car("Mary");
mary.PrintState();

// Создать объект Car по имени Daisy со скоростью 75 миль в час.
Car daisy = new Car("Daisy", 75);
daisy.PrintState();
...
```

Конструкторы в виде членов, сжатых до выражений (нововведение в версии 7.0)

В C# 7 появились дополнительные случаи употребления для стиля членов, сжатых до выражений. Теперь такой синтаксис применим к конструкторам, финализаторам, а также к средствам доступа `get/set` для свойств и индексаторов. С учетом сказанного предыдущий конструктор можно переписать следующим образом:

```
// Здесь currSpeed получит стандартное
// значение для типа int (0).
public Car(string pn) => petName = pn;
```

Второй специальный конструктор не может быть преобразован в выражение, т.к. члены, сжатые до выражений, должны быть однострочными методами.

Конструкторы с параметрами `out` (нововведение в версии 7.3)

Начиная с версии C# 7.3, в конструкторах (а также в рассматриваемых позже инициализаторах полей и свойств) могут использоваться параметры `out`. В качестве простого примера добавьте в класс `Car` следующий конструктор:

```
public Car(string pn, int cs, out bool inDanger)
{
  petName = pn;
  currSpeed = cs;
  if (cs > 100)
  {
    inDanger = true;
  }
  else
  {
    inDanger = false;
  }
}
```

Как обычно, должны соблюдаться все правила, касающиеся параметров `out`. В приведенном примере параметру `inDanger` потребуется присвоить значение до завершения конструктора.

Еще раз о стандартном конструкторе

Как вы только что узнали, все классы снабжаются стандартным конструктором. Добавьте в свой проект новый файл по имени `Motorcycle.cs` с показанным ниже определением класса `Motorcycle`:

```
using System;
namespace SimpleClassExample
{
  class Motorcycle
  {
    public void PopAWheely()
    {
      Console.WriteLine("Yeeeeeee Haaaaaeewww!");
    }
  }
}
```

Теперь появилась возможность создания экземпляров `Motorcycle` с помощью стандартного конструктора:

```
Console.WriteLine("***** Fun with Class Types *****\n");
Motorcycle mc = new Motorcycle();
mc.PopAWheely();
...
```

Тем не менее, как только определен специальный конструктор с любым числом параметров, стандартный конструктор молча удаляется из класса и перестает быть доступным. Воспринимайте это так: если вы не определили специальный конструктор, тогда компилятор C# снабжает класс стандартным конструктором, давая возможность пользователю размещать в памяти экземпляр вашего класса с набором полей данных, которые установлены в корректные стандартные значения. Однако когда вы определяете уникальный конструктор, то компилятор предполагает, что вы решили взять власть в свои руки.

Следовательно, если вы хотите позволить пользователю создавать экземпляр вашего типа с помощью стандартного конструктора, а также специального конструктора, то должны *явно* переопределить стандартный конструктор. Важно понимать, что в подавляющем большинстве случаев реализация стандартного конструктора класса намеренно оставляется пустой, т.к. требуется только создание объекта со стандартными значениями. Обновите класс `Motorcycle`:

```
class Motorcycle
{
  public int driverIntensity;

  public void PopAWheely()
  {
    for (int i = 0; i <= driverIntensity; i++)
    {
      Console.WriteLine("Yeeeeeee Haaaaaeewww!");
    }
  }

  // Вернуть стандартный конструктор, который будет
  // устанавливать все члены данных в стандартные значения.
```

```
  public Motorcycle() {}

  // Специальный конструктор.
  public Motorcycle(int intensity)
  {
    driverIntensity = intensity;
  }
}
```

На заметку! Теперь, когда вы лучше понимаете роль конструкторов класса, полезно узнать об одном удобном сокращении. В Visual Studio и Visual Studio Code предлагается фрагмент кода `ctor`. Если вы наберете `ctor` и нажмете клавишу <Tab>, тогда IDE-среда автоматически определит специальный стандартный конструктор. Затем можно добавить нужные параметры и логику реализации. Испытайте такой прием.

Роль ключевого слова `this`

В языке C# имеется ключевое слово `this`, которое обеспечивает доступ к текущему экземпляру класса. Один из возможных сценариев использования `this` предусматривает устранение неоднозначности с областью видимости, которая может возникнуть, когда входной параметр имеет такое же имя, как и поле данных класса. Разумеется, вы могли бы просто придерживаться соглашения об именовании, которое не приводит к такой неоднозначности; тем не менее, чтобы проиллюстрировать такой сценарий, добавьте в класс `Motorcycle` новое поле типа `string` (под названием `name`), предназначенное для представления имени водителя. Затем добавьте метод `SetDriverName()` со следующей реализацией:

```
class Motorcycle
{
  public int driverIntensity;

  // Новые члены для представления имени водителя.
  public string name;
  public void SetDriverName(string name) => name = name;
  ...
}
```

Хотя приведенный код нормально скомпилируется, компилятор C# выдаст сообщение с предупреждением о том, что переменная присваивается сама себе! В целях иллюстрации добавьте в свой код вызов метода `SetDriverName()` и обеспечьте вывод значения поля `name`. Вы можете быть удивлены, обнаружив, что значением поля `name` является пустая строка!

```
// Создать объект Motorcycle с мотоциклистом по имени Tiny?
Motorcycle c = new Motorcycle(5);
c.SetDriverName("Tiny");
c.PopAWheely();
Console.WriteLine("Rider name is {0}", c.name);
                  // Выводит пустое значение name!
```

Проблема в том, что реализация метода `SetDriverName()` присваивает входному параметру значение *его самого*, т.к. компилятор предполагает, что `name` ссылается на переменную, находящуюся в области видимости метода, а не на поле `name` из области видимости класса. Для информирования компилятора о том, что необходимо уста-

новить поле данных name текущего объекта в значение входного параметра name, просто используйте ключевое слово this, устранив такую неоднозначность:

```
public void SetDriverName(string name) => this.name = name;
```

Если неоднозначность отсутствует, тогда применять ключевое слово this для доступа класса к собственным полям данных или членам вовсе не обязательно. Например, если вы переименуете член данных типа string с name на driverName (что также повлечет за собой модификацию операторов верхнего уровня), то потребность в использовании this отпадет, поскольку неоднозначности с областью видимости больше нет:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  public void SetDriverName(string name)
  {
    // Эти два оператора функционально эквивалентны.
    driverName = name;
    this.driverName = name;
  }
  ...
}
```

Несмотря на то что применение ключевого слова this в неоднозначных ситуациях дает не особенно большой выигрыш, вы можете счесть его удобным при реализации членов класса, т.к. IDE-среды, подобные Visual Studio и Visual Studio Code, будут активизировать средство IntelliSense, когда присутствует this. Это может оказаться полезным, если вы забыли имя члена класса и хотите быстро вспомнить его определение.

На заметку! Общепринятое соглашение об именовании предусматривает снабжение имен закрытых (или внутренних) переменных уровня класса префиксом в виде символа подчеркивания (скажем, _driverName), чтобы средство IntelliSense отображало все ваши переменные в верхней части списка. В нашем простом примере все поля являются открытыми, поэтому такое соглашение об именовании не применяется. В остальном материале книги закрытые и внутренние переменные будут именоваться с ведущим символом подчеркивания.

Построение цепочки вызовов конструкторов с использованием this

Еще один сценарий применения ключевого слова this касается проектирования класса с использованием приема, который называется *построением цепочки конструкторов*. Такой паттерн проектирования полезен при наличии класса, определяющего множество конструкторов. Учитывая тот факт, что конструкторы нередко проверяют входные аргументы на предмет соблюдения разнообразных бизнес-правил, довольно часто внутри набора конструкторов обнаруживается избыточная логика проверки достоверности. Рассмотрим следующее модифицированное определение класса Motorcycle:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  public Motorcycle() { }

  // Избыточная логика конструктора!
  public Motorcycle(int intensity)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
  }

  public Motorcycle(int intensity, string name)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

Здесь (возможно в попытке обеспечить безопасность мотоциклиста) внутри каждого конструктора производится проверка того, что уровень мощности не превышает значения 10. Наряду с тем, что это правильно, в двух конструкторах присутствует избыточный код. Подход далек от идеала, поскольку в случае изменения правил (например, если уровень мощности не должен превышать значение 5 вместо 10) код придется модифицировать в нескольких местах.

Один из способов улучшить создавшуюся ситуацию предусматривает определение в классе `Motorcycle` метода, который будет выполнять проверку входных аргументов. Если вы решите поступить так, тогда каждый конструктор сможет вызывать такой метод перед присваиванием значений полям. Хотя описанный подход позволяет изолировать код, который придется обновлять при изменении бизнес-правил, теперь появилась другая избыточность:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  // Конструкторы.
  public Motorcycle() { }

  public Motorcycle(int intensity)
  {
    SetIntensity(intensity);
  }

  public Motorcycle(int intensity, string name)
  {
```

```
    SetIntensity(intensity);
    driverName = name;
  }
  public void SetIntensity(int intensity)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
  }
  ...
}
```

Более совершенный подход предполагает назначение конструктора, который принимает *наибольшее количество аргументов*, в качестве "главного конструктора" и выполнение требуемой логики проверки достоверности внутри его реализации. Остальные конструкторы могут применять ключевое слово `this` для передачи входных аргументов главному конструктору и при необходимости предоставлять любые дополнительные параметры. В таком случае вам придется беспокоиться только о поддержке единственного конструктора для всего класса, в то время как оставшиеся конструкторы будут в основном пустыми.

Ниже представлена финальная реализация класса `Motorcycle` (с одним дополнительным конструктором в целях иллюстрации). При связывании конструкторов в цепочку обратите внимание, что ключевое слово `this` располагается за пределами самого конструктора и отделяется от его объявления двоеточием:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;
  // Связывание конструкторов в цепочку.
  public Motorcycle() {}
  public Motorcycle(int intensity)
    : this(intensity, "") {}
  public Motorcycle(string name)
    : this(0, name) {}
  // Это 'главный' конструктор, выполняющий всю реальную работу.
  public Motorcycle(int intensity, string name)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

Имейте в виду, что использовать ключевое слово `this` для связывания вызовов конструкторов в цепочку вовсе не обязательно. Однако такой подход позволяет получить лучше сопровождаемое и более краткое определение класса. Применяя данный прием, также можно упростить решение задач программирования, потому что

реальная работа делегируется единственному конструктору (обычно принимающему большую часть параметров), тогда как остальные просто "перекладывают на него ответственность".

На заметку! Вспомните из главы 4, что в языке C# поддерживаются необязательные параметры. Если вы будете использовать в конструкторах своих классов необязательные параметры, то сможете добиться тех же преимуществ, что и при связывании конструкторов в цепочку, но с меньшим объемом кода. Вскоре вы увидите, как это делается.

Исследование потока управления конструкторов

Напоследок отметим, что как только конструктор передал аргументы выделенному главному конструктору (и главный конструктор обработал данные), первоначально вызванный конструктор продолжит выполнение всех оставшихся операторов кода. В целях прояснения модифицируйте конструкторы класса `Motorcycle`, добавив в них вызов метода `Console.WriteLine()`:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;
  // Связывание конструкторов в цепочку.
  public Motorcycle()
  {
    Console.WriteLine("In default ctor");
                    // Внутри стандартного конструктора
  }
  public Motorcycle(int intensity)
     : this(intensity, "")
  {
    Console.WriteLine("In ctor taking an int");
                    // Внутри конструктора, принимающего int
  }
  public Motorcycle(string name)
     : this(0, name)
  {
    Console.WriteLine("In ctor taking a string");
                    // Внутри конструктора, принимающего string
  }
  // Это 'главный' конструктор, выполняющий всю реальную работу.
  public Motorcycle(int intensity, string name)
  {
    Console.WriteLine("In master ctor ");
                    // Внутри главного конструктора
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

Теперь измените операторы верхнего уровня, чтобы они работали с объектом Motorcycle:

```
Console.WriteLine("***** Fun with Motorcycles *****\n");

// Создать объект Motorcycle.
Motorcycle c = new Motorcycle(5);
c.SetDriverName("Tiny");
c.PopAWheely();
Console.WriteLine("Rider name is {0}", c.driverName);
                // вывод имени гонщика
Console.ReadLine();
```

Вот вывод, полученный в результате выполнения показанного выше кода:

```
***** Fun with Motorcycles *****

In master ctor
In ctor taking an int
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Rider name is Tiny
```

Ниже описан поток логики конструкторов.

- Первым делом создается объект путем вызова конструктора, принимающего один аргумент типа int.
- Этот конструктор передает полученные данные главному конструктору и предоставляет любые дополнительные начальные аргументы, не указанные вызывающим кодом.
- Главный конструктор присваивает входные данные полям данных объекта.
- Управление возвращается первоначально вызванному конструктору, который выполняет оставшиеся операторы кода.

В построении цепочек конструкторов примечательно то, что данный шаблон программирования будет работать с любой версией языка C# и платформой .NET Core. Тем не менее, если целевой платформой является .NET 4.0 или последующая версия, то решение задач можно дополнительно упростить, применяя необязательные аргументы в качестве альтернативы построению традиционных цепочек конструкторов.

Еще раз о необязательных аргументах

В главе 4 вы изучили необязательные и именованные аргументы. Вспомните, что необязательные аргументы позволяют определять стандартные значения для входных аргументов. Если вызывающий код устраивают стандартные значения, то указывать уникальные значения не обязательно, но это нужно делать, чтобы снабдить объект специальными данными. Рассмотрим следующую версию класса Motorcycle, которая теперь предлагает несколько возможностей конструирования объектов, используя *единственное* определение конструктора:

```
class Motorcycle
{
  // Единственный конструктор, использующий необязательные аргументы.
  public Motorcycle(int intensity = 0, string name = "")
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

С помощью такого единственного конструктора можно создавать объект `Motorcycle`, указывая ноль, один или два аргумента. Вспомните, что синтаксис именованных аргументов по существу позволяет пропускать подходящие стандартные установки (см. главу 4).

```
static void MakeSomeBikes()
{
  // driverName = "", driverIntensity = 0
  Motorcycle m1 = new Motorcycle();
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m1.driverName, m1.driverIntensity);

  // driverName = "Tiny", driverIntensity = 0
  Motorcycle m2 = new Motorcycle(name:"Tiny");
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m2.driverName, m2.driverIntensity);

  // driverName = "", driverIntensity = 7
  Motorcycle m3 = new Motorcycle(7);
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m3.driverName, m3.driverIntensity);
}
```

В любом случае к настоящему моменту вы способны определить класс с полями данных (т.е. переменными-членами) и разнообразными операциями, такими как методы и конструкторы. А теперь формализуем роль ключевого слова `static`.

Понятие ключевого слова `static`

В классе C# можно определять любое количество *статических членов*, объявляемых с применением ключевого слова `static`. В таком случае интересующий член должен вызываться прямо на уровне класса, а не через переменную со ссылкой на объект. Чтобы проиллюстрировать разницу, обратимся к нашему старому знакомому классу `System.Console`. Как вы уже видели, метод `WriteLine()` не вызывается на уровне объекта:

```
// Ошибка на этапе компиляции! WriteLine() - не метод уровня объекта!
Console c = new Console();
c.WriteLine("I can't be printed...");
```

Взамен статический член `WriteLine()` предваряется именем класса:

```
// Правильно! WriteLine() - статический метод.
Console.WriteLine("Much better! Thanks...");
```

Выражаясь просто, статические члены — это элементы, которые проектировщик класса посчитал настолько общими, что перед обращением к ним даже нет нужды создавать экземпляр класса. Наряду с тем, что определять статические члены можно в любом классе, чаще всего они обнаруживаются внутри *обслуживающих классов*. По определению обслуживающий класс представляет собой такой класс, который не поддерживает какое-либо состояние на уровне объектов и не предполагает создание своих экземпляров с помощью ключевого слова `new`. Взамен обслуживающий класс открывает доступ ко всей функциональности посредством членов уровня класса (также известных под названием статических).

Например, если бы вы воспользовались браузером объектов Visual Studio (выбрав пункт меню View⇨Object Browser (Вид⇨Браузер объектов)) для просмотра пространства имен `System`, то увидели бы, что все члены классов `Console`, `Math`, `Environment` и `GC` (среди прочих) открывают доступ к своей функциональности через статические члены. Они являются лишь несколькими обслуживающими классами, которые можно найти в библиотеках базовых классов .NET Core.

И снова следует отметить, что статические члены находятся не только в обслуживающих классах; они могут быть частью в принципе любого определения класса. Просто запомните, что статические члены продвигают отдельный элемент на уровень класса вместо уровня объектов. Как будет показано в нескольких последующих разделах, ключевое слово `static` может применяться к перечисленным ниже конструкциям:

- данные класса;
- методы класса;
- свойства класса;
- конструктор;
- полное определение класса;
- в сочетании с ключевым словом `using`.

Давайте рассмотрим все варианты, начав с концепции статических данных.

На заметку! Роль статических свойств будет объясняться позже в главе во время исследования самих свойств.

Определение статических полей данных

При проектировании класса в большинстве случаев данные определяются на уровне экземпляра — другими словами, как нестатические данные. Когда определяются данные уровня экземпляра, то известно, что каждый создаваемый новый объект поддерживает собственную независимую копию этих данных. По контрасту при определении *статических* данных класса выделенная под них память разделяется всеми объектами этой категории.

Чтобы увидеть разницу, создайте новый проект консольного приложения под названием `StaticDataAndMembers`. Добавьте в проект файл по имени `SavingsAccount.cs`

и создайте в нем класс SavingsAccount. Начните с определения переменной уровня экземпляра (для моделирования текущего баланса) и специального конструктора для установки начального баланса:

```
using System;
namespace StaticDataAndMembers
{
  // Простой класс депозитного счета.
  class SavingsAccount
  {
    // Данные уровня экземпляра.
    public double currBalance;

    public SavingsAccount(double balance)
    {
      currBalance = balance;
    }
  }
}
```

При создании объектов SavingsAccount память под поле currBalance выделяется для каждого объекта. Таким образом, можно было бы создать пять разных объектов SavingsAccount, каждый с собственным уникальным балансом. Более того, в случае изменения баланса в одном объекте счета другие объекты не затрагиваются.

С другой стороны, память под статические данные распределяется один раз и используется всеми объектами того же самого класса. Добавьте в класс SavingsAccount статическую переменную по имени currInterestRate, которая устанавливается в стандартное значение 0.04:

```
// Простой класс депозитного счета.
class SavingsAccount
{
  // Статический элемент данных.
  public static double currInterestRate = 0.04;

  // Данные уровня экземпляра.
  public double currBalance;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
}
```

Создайте три экземпляра класса SavingsAccount, как показано ниже:

```
using System;
using StaticDataAndMembers;

Console.WriteLine("***** Fun with Static Data *****\n");
SavingsAccount s1 = new SavingsAccount(50);
SavingsAccount s2 = new SavingsAccount(100);
SavingsAccount s3 = new SavingsAccount(10000.75);
Console.ReadLine();
```

Размещение данных в памяти будет выглядеть примерно так, как иллюстрируется на рис. 5.1.

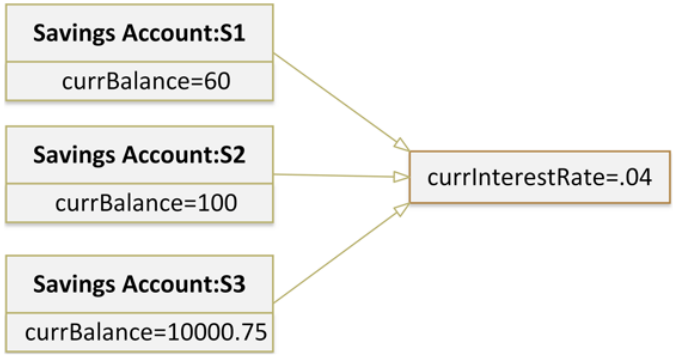

Рис. 5.1. Память под статические данные распределяется один раз и совместно используется всеми экземплярами класса

Здесь предполагается, что все депозитные счета должны иметь одну и ту же процентную ставку. Поскольку статические данные разделяются всеми объектами той же самой категории, если вы измените процентную ставку каким-либо образом, тогда все объекты будут "видеть" новое значение при следующем доступе к статическим данным, т.к. все они по существу просматривают одну и ту же ячейку памяти. Чтобы понять, как изменять (или получать) статические данные, понадобится рассмотреть роль статических методов.

Определение статических методов

Модифицируйте класс `SavingsAccount` с целью определения в нем двух статических методов. Первый статический метод (`GetInterestRate()`) будет возвращать текущую процентную ставку, а второй (`SetInterestRate()`) позволит изменять эту процентную ставку:

```
// Простой класс депозитного счета.
class SavingsAccount
{
  // Данные уровня экземпляра.
  public double currBalance;

  // Статический элемент данных.
  public static double currInterestRate = 0.04;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }

  // Статические члены для установки/получения процентной ставки.
  public static void SetInterestRate(double newRate)
    => currInterestRate = newRate;

  public static double GetInterestRate()
    => currInterestRate;
}
```

Рассмотрим показанный ниже сценарий использования класса:

```
using System;
using StaticDataAndMembers;
Console.WriteLine("***** Fun with Static Data *****\n");
SavingsAccount s1 = new SavingsAccount(50);
SavingsAccount s2 = new SavingsAccount(100);

// Вывести текущую процентную ставку.
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.
GetInterestRate());

// Создать новый объект; это не 'сбросит' процентную ставку.
SavingsAccount s3 = new SavingsAccount(10000.75);
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.
GetInterestRate());

Console.ReadLine();
```

Вывод предыдущего кода выглядит так:

```
***** Fun with Static Data *****
Interest Rate is: 0.04
Interest Rate is: 0.04
```

Как видите, при создании новых экземпляров класса `SavingsAccount` значение статических данных не сбрасывается, поскольку среда CoreCLR выделяет для них место в памяти только один раз. Затем все объекты типа `SavingsAccount` имеют дело с одним и тем же значением в статическом поле `currInterestRate`.

Когда проектируется любой класс C#, одна из задач связана с выяснением того, какие порции данных должны быть определены как статические члены, а какие — нет. Хотя строгих правил не существует, запомните, что поле статических данных разделяется между всеми объектами конкретного класса. Поэтому, если необходимо, чтобы часть данных совместно использовалась *всеми* объектами, то статические члены будут самым подходящим вариантом.

Посмотрим, что произойдет, если поле `currInterestRate` *не* определено с ключевым словом `static`. Это означает, что каждый объект `SavingAccount` будет иметь собственную копию поля `currInterestRate`. Предположим, что вы создали сто объектов `SavingAccount` и нуждаетесь в изменении размера процентной ставки. Такое действие потребовало бы вызова метода `SetInterestRate()` сто раз! Ясно, что подобный способ моделирования "разделяемых данных" трудно считать удобным. Статические данные безупречны в ситуации, когда есть значение, которое должно быть общим для всех объектов заданной категории.

На заметку! Ссылка на нестатические члены внутри реализации статического члена приводит к ошибке на этапе компиляции. В качестве связанного замечания: ошибкой также будет применение ключевого слова `this` к статическому члену, потому что `this` подразумевает объект!

Определение статических конструкторов

Типичный конструктор используется для установки значений данных уровня экземпляра во время его создания. Однако что произойдет, если вы попытаетесь присвоить значение статическому элементу данных в типичном конструкторе? Вы можете быть удивлены, обнаружив, что значение сбрасывается каждый раз, когда создается новый объект.

В целях иллюстрации модифицируйте код конструктора класса SavingsAccount, как показано ниже (также обратите внимание, что поле currInterestRate больше не устанавливается при объявлении):

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate;

  // Обратите внимание, что наш конструктор устанавливает
  // значение статического поля currInterestRate.
  public SavingsAccount(double balance)
  {
    currInterestRate = 0.04;  // Это статические данные!
    currBalance = balance;
  }
  ...
}
```

Теперь добавьте к операторам верхнего уровня следующий код:

```
Console.WriteLine("***** Fun with Static Data *****\n");

// Создать объект счета.
SavingsAccount s1 = new SavingsAccount(50);

// Вывести текущую процентную ставку.
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.
GetInterestRate());

// Попытаться изменить процентную ставку через свойство.
SavingsAccount.SetInterestRate(0.08);

// Создать второй объект счета.
SavingsAccount s2 = new SavingsAccount(100);

// Должно быть выведено 0.08, не так ли?
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.
GetInterestRate());
Console.ReadLine();
```

При выполнении этого кода вы увидите, что переменная currInterestRate сбрасывается каждый раз, когда создается новый объект SavingsAccount, и она всегда установлена в 0.04. Очевидно, что установка значений статических данных в нормальном конструкторе уровня экземпляра сводит на нет все их предназначение. Когда бы ни создавался новый объект, данные уровня класса сбрасываются! Один из подходов к установке статического поля предполагает применение синтаксиса инициализации членов, как делалось изначально:

```
class SavingsAccount
{
  public double currBalance;

  // Статические данные.
  public static double currInterestRate = 0.04;
  ...
}
```

Такой подход обеспечит установку статического поля только один раз независимо от того, сколько объектов создается. Но что, если значение статических данных необходимо получать во время выполнения? Например, в типичном банковском приложении значение переменной, представляющей процентную ставку, будет читаться из базы данных или внешнего файла. Решение задач подобного рода обычно требует области действия метода, такого как конструктор, для выполнения соответствующих операторов кода.

По этой причине язык C# позволяет определять статический конструктор, который дает возможность безопасно устанавливать значения статических данных. Взгляните на следующее изменение в коде класса:

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }

  // Статический конструктор!
  static SavingsAccount()
  {
    Console.WriteLine("In static ctor!");
                    // В статическом конструкторе
    currInterestRate = 0.04;
  }
  ...
}
```

Выражаясь просто, статический конструктор представляет собой специальный конструктор, который является идеальным местом для инициализации значений статических данных, если их значения не известны на этапе компиляции (например, когда значения нужно прочитать из внешнего файла или базы данных, сгенерировать случайные числа либо получить значения еще каким-нибудь способом). Если вы снова запустите предыдущий код, то увидите ожидаемый вывод. Обратите внимание, что сообщение "In static ctor!" выводится только один раз, т.к. среда CoreCLR вызывает все статические конструкторы перед первым использованием (и никогда не вызывает их заново для данного экземпляра приложения):

```
***** Fun with Static Data *****

In static ctor!
Interest Rate is: 0.04
Interest Rate is: 0.08
```

Ниже отмечено несколько интересных моментов, касающихся статических конструкторов.

- В отдельно взятом классе может быть определен только один статический конструктор. Другими словами, перегружать статический конструктор нельзя.
- Статический конструктор не имеет модификатора доступа и не может принимать параметры.

- Статический конструктор выполняется только один раз вне зависимости от количества создаваемых объектов заданного класса.
- Исполняющая система вызывает статический конструктор, когда создает экземпляр класса или перед доступом к первому статическому члену из вызывающего кода.
- Статический конструктор выполняется перед любым конструктором уровня экземпляра.

С учетом такой модификации при создании новых объектов `SavingsAccount` значения статических данных предохраняются, поскольку статический член устанавливается только один раз внутри статического конструктора независимо от количества созданных объектов.

Определение статических классов

Ключевое слово `static` допускается также применять прямо на уровне класса. Когда класс определен как статический, его экземпляры нельзя создавать с использованием ключевого слова `new`, и он может содержать только члены или поля данных, помеченные ключевым словом `static`. В случае нарушения этого правила возникают ошибки на этапе компиляции.

На заметку! Вспомните, что класс (или структура), который открывает доступ только к статической функциональности, часто называется обслуживающим классом. При проектировании обслуживающего класса рекомендуется применять ключевое слово `static` к самому определению класса.

На первый взгляд такое средство может показаться довольно странным, учитывая невозможность создания экземпляров класса. Тем не менее, в первую очередь класс, который содержит только статические члены и/или константные данные, не нуждается в выделении для него памяти. В целях иллюстрации определите новый класс по имени `TimeUtilClass`:

```
using System;
namespace StaticDataAndMembers
{
  // Статические классы могут содержать только статические члены!
  static class TimeUtilClass
  {
    public static void PrintTime()
      => Console.WriteLine(DateTime.Now.ToShortTimeString());
    public static void PrintDate()
      => Console.WriteLine(DateTime.Today.ToShortDateString());
  }
}
```

Так как класс `TimeUtilClass` определен с ключевым словом `static`, создавать его экземпляры с помощью ключевого слова `new` нельзя. Взамен вся функциональность доступна на уровне класса. Чтобы протестировать данный класс, добавьте к операторам верхнего уровня следующий код:

```
Console.WriteLine("***** Fun with Static Classes *****\n");
// Это работает нормально.
TimeUtilClass.PrintDate();
TimeUtilClass.PrintTime();
// Ошибка на этапе компиляции!
// Создавать экземпляры статического класса невозможно!
TimeUtilClass u = new TimeUtilClass ();
Console.ReadLine();
```

Импортирование статических членов с применением ключевого слова `using` языка C#

В версии C# 6 появилась поддержка импортирования статических членов с помощью ключевого слова `using`. В качестве примера предположим, что в файле C# определен обслуживающий класс. Поскольку в нем делаются вызовы метода `WriteLine()` класса `Console`, а также обращения к свойствам `Now` и `Today` класса `DateTime`, должен быть предусмотрен оператор `using` для пространства имен `System`. Из-за того, что все члены упомянутых классов являются статическими, в файле кода можно указать следующие директивы `using static`:

```
// Импортировать статические члены классов Console и DateTime.
using static System.Console;
using static System.DateTime;
```

После такого "статического импортирования" в файле кода появляется возможность напрямую применять статические методы классов `Console` и `DateTime`, не снабжая их префиксом в виде имени класса, в котором они определены. Например, модифицируем наш обслуживающий класс `TimeUtilClass`, как показано ниже:

```
static class TimeUtilClass
{
  public static void PrintTime()
    => WriteLine(Now.ToShortTimeString());
  public static void PrintDate()
    => WriteLine(Today.ToShortDateString());
}
```

В более реалистичном примере упрощения кода за счет импортирования статических членов мог бы участвовать класс C#, интенсивно использующий класс `System.Math` (или какой-то другой обслуживающий класс). Поскольку этот класс содержит только статические члены, отчасти было бы проще указать для него оператор `using static` и затем напрямую обращаться членам класса `Math` в своем файле кода.

Однако имейте в виду, что злоупотребление операторами статического импортирования может привести в результате к путанице. Во-первых, как быть, если метод `WriteLine()` определен сразу в нескольких классах? Будет сбит с толку как компилятор, так и другие программисты, читающие ваш код. Во-вторых, если разработчик не особенно хорошо знаком с библиотеками кода .NET Core, то он может не знать о том, что `WriteLine()` является членом класса `Console`. До тех пор, пока разработчик не заметит набор операторов статического импортирования в начале файла кода C#, он не может быть полностью уверен в том, где данный метод фактически определен. По указанным причинам применение операторов `using static` в книге ограничено.

К настоящему моменту вы должны уметь определять простые типы классов, содержащие конструкторы, поля и разнообразные статические (и нестатические) члены. Обладая такими базовыми знаниями о конструкции классов, можно приступать к ознакомлению с тремя основными принципами объектно-ориентированного программирования (ООП).

Основные принципы объектно-ориентированного программирования

Все объектно-ориентированные языки (C#, Java, C++, Visual Basic и т.д.) должны поддерживать три основных принципа ООП.

- **Инкапсуляция.** Каким образом язык скрывает детали внутренней реализации объектов и предохраняет целостность данных?
- **Наследование.** Каким образом язык стимулирует многократное использование кода?
- **Полиморфизм.** Каким образом язык позволяет трактовать связанные объекты в сходной манере?

Прежде чем погрузиться в синтаксические детали каждого принципа, важно понять их базовые роли. Ниже предлагается обзор всех принципов, а в оставшейся части этой и в следующей главе приведены подробные сведения, связанные с ними.

Роль инкапсуляции

Первый основной принцип ООП называется *инкапсуляцией*. Такая характерная черта описывает способность языка скрывать излишние детали реализации от пользователя объекта. Например, предположим, что вы имеете дело с классом по имени `DaabaseReader`, в котором определены два главных метода: `Open()` и `Close()`.

```
// Пусть этот класс инкапсулирует детали открытия и закрытия базы данных.
DatabaseReader dbReader = new DatabaseReader();
dbReader.Open(@"C:\AutoLot.mdf");

// Сделать что-то с файлом данных и закрыть файл.
dbReader.Close();
```

Вымышленный класс `DatabaseReader` инкапсулирует внутренние детали нахождения, загрузки, манипулирования и закрытия файла данных. Программистам нравится инкапсуляция, т.к. этот основной принцип ООП упрощает задачи кодирования. Отсутствует необходимость беспокоиться о многочисленных строках кода, которые работают "за кулисами", чтобы обеспечить функционирование класса `DatabaseReader`. Все, что понадобится — создать экземпляр и отправить ему подходящие сообщения (например, открыть файл по имени `AutoLot.mdf`, расположенный на диске C:).

С понятием инкапсуляции программной логики тесно связана идея защиты данных. В идеале данные состояния объекта должны быть определены с применением одного из ключевых слов `private`, `internal` или `protected`. В итоге внешний мир должен вежливо попросить об изменении либо извлечении лежащего в основе значения, что крайне важно, т.к. открыто объявленные элементы данных легко могут стать поврежденными (конечно, лучше случайно, чем намеренно). Вскоре будет дано формальное определение такого аспекта инкапсуляции.

Роль наследования

Следующий принцип ООП — *наследование* — отражает возможность языка разрешать построение определений новых классов на основе определений существующих классов. По сути, наследование позволяет расширять поведение базового (или *родительского*) класса за счет наследования его основной функциональности производным подклассом (также называемым *дочерним* классом). На рис. 5.2 показан простой пример.

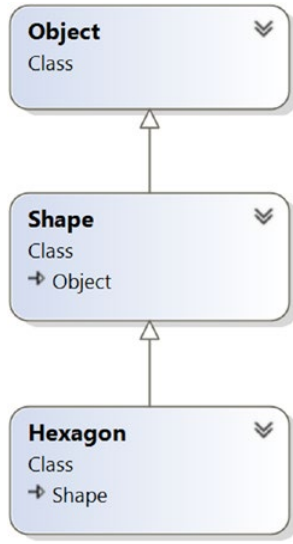

Рис. 5.2. Отношение "является"

Диаграмма на рис. 5.2 читается так: "шестиугольник (`Hexagon`) является фигурой (`Shape`), которая является объектом (`Object`)". При наличии классов, связанных такой формой наследования, между типами устанавливается *отношение "является" ("is-a")*. Отношение "является" называется *наследованием*.

Здесь можно предположить, что класс `Shape` определяет некоторое количество членов, являющихся общими для всех наследников (скажем, значение для представления цвета фигуры, а также значения для высоты и ширины). Учитывая, что класс `Hexagon` расширяет `Shape`, он наследует основную функциональность, определяемую классами `Shape` и `Object`, и вдобавок сам определяет дополнительные детали, связанные с шестиугольником (какими бы они ни были).

На заметку! В рамках платформ .NET/.NET Core класс `System.Object` всегда находится на вершине любой иерархии классов, являясь первоначальным родительским классом, и определяет общую функциональность для всех типов (как подробно объясняется в главе 6).

В мире ООП существует еще одна форма повторного использования кода: модель включения/делегации, также известная как *отношение "имеет" ("has-a")* или *агрегация*. Такая форма повторного использования не применяется для установки отношений "родительский–дочерний". На самом деле отношение "имеет" позволяет одному классу определять переменную-член другого класса и опосредованно (когда требуется) открывать доступ к его функциональности пользователю объекта.

Например, предположим, что снова моделируется автомобиль. Может возникнуть необходимость выразить идею, что автомобиль "имеет" радиоприемник. Было бы нелогично пытаться наследовать класс `Car` (автомобиль) от класса `Radio` (радиоприемник) или наоборот (ведь `Car` не "является" `Radio`). Взамен есть два независимых класса, работающих совместно, где класс `Car` создает и открывает доступ к функциональности класса `Radio`:

```
class Radio
{
  public void Power(bool turnOn)
  {
    Console.WriteLine("Radio on: {0}", turnOn);
  }
}
```

```
class Car
{
  // Car 'имеет' Radio.
  private Radio myRadio = new Radio();

  public void TurnOnRadio(bool onOff)
  {
    // Делегировать вызов внутреннему объекту.
    myRadio.Power(onOff);
  }
}
```

Обратите внимание, что пользователю объекта ничего не известно об использовании классом `Car` внутреннего объекта `Radio`:

```
// Внутренне вызов передается объекту Radio.
Car viper = new Car();
viper.TurnOnRadio(false);
```

Роль полиморфизма

Последним основным принципом ООП является *полиморфизм*. Указанная характерная черта обозначает способность языка трактовать связанные объекты в сходной манере. В частности, данный принцип ООП позволяет базовому классу определять набор членов (формально называемый *полиморфным интерфейсом*), которые доступны всем наследникам. Полиморфный интерфейс класса конструируется с применением любого количества *виртуальных* или *абстрактных* членов (подробности ищите в главе 6).

Выражаясь кратко, *виртуальный член* — это член базового класса, определяющий стандартную реализацию, которую можно изменять (или более формально *переопределять*) в производном классе. В отличие от него *абстрактный метод* — это член базового класса, который не предоставляет стандартную реализацию, а предлагает только сигнатуру. Если класс унаследован от базового класса, в котором определен абстрактный метод, то такой метод *должен* быть переопределен в производном классе. В любом случае, когда производные классы переопределяют члены, определенные в базовом классе, по существу они переопределяют свою реакцию на тот же самый запрос.

Чтобы увидеть полиморфизм в действии, давайте предоставим некоторые детали иерархии фигур, показанной на рис. 5.3. Предположим, что в классе `Shape` определен виртуальный метод `Draw()`, не принимающий параметров. С учетом того, что каждой фигуре необходимо визуализировать себя уникальным образом, подклассы вроде `Hexagon` и `Circle` могут переопределять метод `Draw()` по своему усмотрению (см. рис. 5.3).

После того как полиморфный интерфейс спроектирован, можно начинать делать разнообразные предположения в коде. Например, так как классы `Hexagon` и `Circle` унаследованы от общего родителя (`Shape`), массив элементов типа `Shape` может содержать любые объекты классов, производных от этого базового класса. Более того, поскольку класс `Shape` определяет полиморфный интерфейс для всех производных типов (метод `Draw()` в данном примере), уместно предположить, что каждый член массива обладает такой функциональностью.

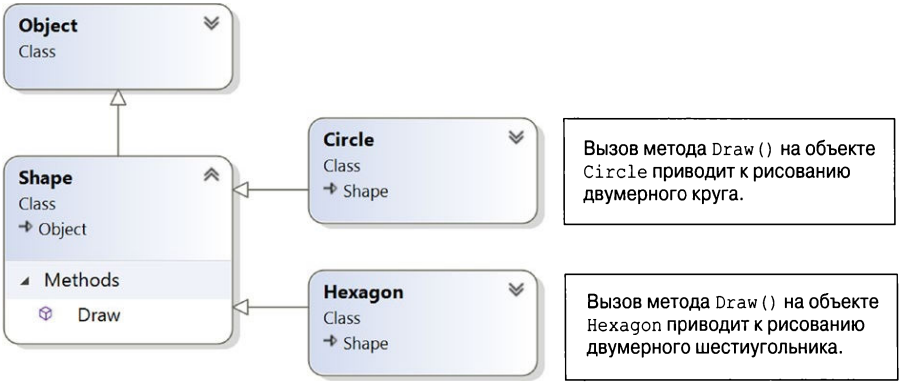

Рис. 5.3. Классический полиморфизм

Рассмотрим следующий код, который заставляет массив элементов производных от `Shape` типов визуализировать себя с использованием метода `Draw()`:

```
Shape[] myShapes = new Shape[3];
myShapes[0] = new Hexagon();
myShapes[1] = new Circle();
myShapes[2] = new Hexagon();
foreach (Shape s in myShapes)
{
  // Использовать полиморфный интерфейс!
  s.Draw();
}
Console.ReadLine();
```

На этом краткий обзор основных принципов ООП завершен. Оставшийся материал главы посвящен дальнейшим подробностям поддержки инкапсуляции в языке C#, начиная с модификаторов доступа. Детали наследования и полиморфизма обсуждаются в главе 6.

Модификаторы доступа C# (обновление в версии 7.2)

При работе с инкапсуляцией вы должны всегда принимать во внимание то, какие аспекты типа являются видимыми различным частям приложения. В частности, типы (классы, интерфейсы, структуры, перечисления и делегаты), а также их члены (свойства, методы, конструкторы и поля) определяются с использованием специального ключевого слова, управляющего "видимостью" элемента для других частей приложения. Хотя в C# для управления доступом предусмотрены многочисленные ключевые слова, они отличаются в том, к чему могут успешно применяться (к типу или члену). Модификаторы доступа и особенности их использования описаны в табл. 5.1.

Таблица 5.1. Модификаторы доступа C#

Модификатор доступа	К чему может быть применен	Практический смысл
`public`	Типы или члены типов	Открытые (`public`) элементы не имеют ограничений доступа. Открытый член может быть доступен из объекта, а также из любого производного класса. Открытый тип может быть доступен из других внешних сборок
`private`	Члены типов или вложенные типы	Закрытые (`private`) элементы могут быть доступны только классу (или структуре), где они определены
`protected`	Члены типов или вложенные типы	Защищенные (`protected`) элементы могут использоваться классом, который их определяет, и любым дочерним классом. Защищенные элементы не доступны за пределами цепочки наследования
`internal`	Типы или члены типов	Внутренние (`internal`) элементы доступны только внутри текущей сборки. Другим сборкам можно явно предоставить разрешение видеть внутренние элементы
`protected internal`	Члены типов или вложенные типы	Когда в объявлении элемента указана комбинация ключевых слов `protected` и `internal`, то такой элемент будет доступен внутри определяющей его сборки, внутри определяющего класса и производным классам внутри или за пределами определяющей сборки
`private protected` (нововведение в версии 7.2)	Члены типов или вложенные типы	Когда в объявлении элемента указана комбинация ключевых слов `private` и `protected`, то такой элемент будет доступен внутри определяющего его класса и производным классам в той же самой сборке

В текущей главе рассматриваются только ключевые слова `public` и `private`. В последующих главах будет исследована роль модификаторов `internal` и `protected internal` (удобных при построении библиотек кода и модульных тестов) и модификатора `protected` (полезного при создании иерархий классов).

Использование стандартных модификаторов доступа

По умолчанию члены типов являются *неявно закрытыми* (`private`), тогда как сами типы — *неявно внутренними* (`internal`). Таким образом, следующее определение класса автоматически устанавливается как `internal`, а стандартный конструктор типа — как `private` (тем не менее, как и можно было предполагать, закрытые конструкторы классов нужны редко):

```
// Внутренний класс с закрытым стандартным конструктором.
class Radio
{
  Radio(){}
}
```

Если вы предпочитаете явное объявление, тогда можете добавить соответствующие ключевые слова без каких-либо негативных последствий (помимо дополнительных усилий по набору):

```
// Внутренний класс с закрытым стандартным конструктором.
internal class Radio
{
  private Radio(){}
}
```

Чтобы позволить другим частям программы обращаться к членам объекта, вы должны определить эти члены с ключевым словом `public` (или возможно с ключевым словом `protected`, которое объясняется в следующей главе). Вдобавок, если вы хотите открыть доступ к `Radio` внешним сборкам (что удобно при построении более крупных решений или библиотек кода), то к нему придется добавить модификатор `public`:

```
// Открытый класс с открытым стандартным конструктором.
public class Radio
{
  public Radio(){}
}
```

Использование модификаторов доступа и вложенных типов

Как упоминалось в табл. 5.1, модификаторы доступа `private`, `protected`, `protected internal` и `private protected` могут применяться к *вложенному типу*. Вложение типов будет подробно рассматриваться в главе 6, а пока достаточно знать, что вложенный тип — это тип, объявленный прямо внутри области видимости класса или структуры. В качестве примера ниже приведено закрытое перечисление (по имени `CarColor`), вложенное в открытый класс (по имени `SportsCar`):

```
public class SportsCar
{
  // Нормально! Вложенные типы могут быть помечены как private.
  private enum CarColor
  {
    Red, Green, Blue
  }
}
```

Здесь допустимо применять модификатор доступа `private` к вложенному типу. Однако невложенные типы (вроде `SportsCar`) могут определяться только с модификатором `public` или `internal`. Таким образом, следующее определение класса незаконно:

```
// Ошибка! Невложенный тип не может быть помечен как private!
private class SportsCar
{}
```

Первый принцип объектно-ориентированного программирования: службы инкапсуляции C#

Концепция инкапсуляции вращается вокруг идеи о том, что данные класса не должны быть напрямую доступными через его экземпляр. Наоборот, данные класса определяются как закрытые. Если пользователь объекта желает изменить его состояние, тогда он должен делать это косвенно, используя открытые члены. Чтобы проиллюстрировать необходимость в службах инкапсуляции, предположим, что создано такое определение класса:

```
// Класс с единственным открытым полем.
class Book
{
  public int numberOfPages;
}
```

Проблема с открытыми данными заключается в том, что сами по себе они неспособны "понять", является ли присваиваемое значение допустимым с точки зрения текущих бизнес-правил системы. Как известно, верхний предел значений для типа `int` в C# довольно высок (2 147 483 647), поэтому компилятор разрешит следующее присваивание:

```
// Хм... Ничего себе мини-новелла!
Book miniNovel = new Book();
miniNovel.numberOfPages = 30_000_000;
```

Хотя границы типа данных `int` не превышены, понятно, что мини-новелла объемом 30 миллионов страниц выглядит несколько неправдоподобно. Как видите, открытые поля не предоставляют способа ограничения значений верхними (или нижними) логическими пределами. Если в системе установлено текущее бизнес-правило, которое регламентирует, что книга должна иметь от 1 до 1000 страниц, то совершенно неясно, как обеспечить его выполнение программным образом. Именно потому открытым полям обычно нет места в определениях классов производственного уровня.

На заметку! Говоря точнее, члены класса, которые представляют состояние объекта, не должны помечаться как `public`. В то же время позже в главе вы увидите, что вполне нормально иметь открытые константы и открытые поля, допускающие только чтение.

Инкапсуляция предлагает способ предохранения целостности данных состояния для объекта. Вместо определения открытых полей (которые могут легко привести к повреждению данных) необходимо выработать у себя привычку определять *закрытые данные*, управление которыми осуществляется опосредованно с применением одного из двух главных приемов:

- определение пары открытых методов доступа и изменения;
- определение открытого свойства.

Независимо от выбранного приема идея заключается в том, что хорошо инкапсулированный класс должен защищать свои данные и скрывать подробности своего функционирования от любопытных глаз из внешнего мира. Это часто называют *программированием в стиле черного ящика*. Преимущество такого подхода в том, что объект может свободно изменять внутреннюю реализацию любого метода. Работа су-

ществующего кода, который использует данный метод, не нарушается при условии, что параметры и возвращаемые значения методов остаются неизменными.

Инкапсуляция с использованием традиционных методов доступа и изменения

В оставшейся части главы будет построен довольно полный класс, моделирующий обычного сотрудника. Для начала создайте новый проект консольного приложения под названием EmployeeApp и добавьте в него новый файл класса по имени Employee.cs. Обновите класс Employee с применением следующего пространства имен, полей, методов и конструкторов:

```
using System;
namespace EmployeeApp
{
  class Employee
  {
    // Поля данных.
    private string _empName;
    private int _empId;
    private float _currPay;

    // Конструкторы.
    public Employee() {}
    public Employee(string name, int id, float pay)
    {
      _empName = name;
      _empId = id;
      _currPay = pay;
    }

    // Методы.
    public void GiveBonus(float amount) => _currPay += amount;
    public void DisplayStats()
    {
      Console.WriteLine("Name: {0}", _empName);   // имя сотрудника
      Console.WriteLine("ID: {0}", _empId);       // идентификационный
                                                  //   номер сотрудника
      Console.WriteLine("Pay: {0}", _currPay);    // текущая выплата
    }
  }
}
```

Обратите внимание, что поля класса Employee в текущий момент определены с использованием ключевого слова private. Учитывая это, поля empName, empID и currPay не будут доступными напрямую через объектную переменную. Таким образом, показанная ниже логика в коде приведет к ошибкам на этапе компиляции:

```
Employee emp = new Employee();
// Ошибка! Невозможно напрямую обращаться к закрытым полям объекта!
emp.empName = "Marv";
```

Если нужно, чтобы внешний мир взаимодействовал с полным именем сотрудника, то традиционный подход предусматривает определение методов доступа (метод get) и изменения (метод set). Роль метода get заключается в возвращении вызывающему

коду текущего значения лежащих в основе данных состояния. Метод `set` позволяет вызывающему коду изменять текущее значение лежащих в основе данных состояния при условии удовлетворения бизнес-правил.

В целях иллюстрации давайте инкапсулируем поле `empName`, для чего к существующему классу `Employee` необходимо добавить показанные ниже открытые методы. Обратите внимание, что метод `SetName()` выполняет проверку входных данных, чтобы удостовериться, что строка имеет длину 15 символов или меньше. Если это не так, тогда на консоль выводится сообщение об ошибке и происходит возврат без внесения изменений в поле `empName`.

На заметку! В случае класса производственного уровня проверку длины строки с именем сотрудника следовало бы предусмотреть также и внутри логики конструктора. Мы пока проигнорируем указанную деталь, но улучшим код позже, во время исследования синтаксиса свойств.

```
class Employee
{
  // Поля данных.
  private string _empName;
  ...
  // Метод доступа (метод get).
  public string GetName() => _empName;
  // Метод изменения (метод set).
  public void SetName(string name)
  {
    // Перед присваиванием проверить входное значение.
    if (name.Length > 15)
    {
      Console.WriteLine("Error! Name length exceeds 15 characters!");
                  // Ошибка! Длина имени превышает 15 символов!
    }
    else
    {
      _empName = name;
    }
  }
}
```

Такой подход требует наличия двух уникально именованных методов для управления единственным элементом данных. Чтобы протестировать новые методы, модифицируйте свой код следующим образом:

```
Console.WriteLine("***** Fun with Encapsulation *****\n");
Employee emp = new Employee("Marvin", 456, 30000);
emp.GiveBonus(1000);
emp.DisplayStats();
// Использовать методы get/set для взаимодействия
// с именем сотрудника, представленного объектом.
emp.SetName("Marv");
Console.WriteLine("Employee is named: {0}", emp.GetName());
Console.ReadLine();
```

Благодаря коду в методе SetName() попытка указать для имени строку, содержащую более 15 символов (как показано ниже), приводит к выводу на консоль жестко закодированного сообщения об ошибке:

```
Console.WriteLine("***** Fun with Encapsulation *****\n");
...
// Длиннее 15 символов! На консоль выводится сообщение об ошибке.
Employee emp2 = new Employee();
emp2.SetName("Xena the warrior princess");
Console.ReadLine();
```

Пока все идет хорошо. Мы инкапсулировали закрытое поле empName с использованием двух открытых методов с именами GetName() и SetName(). Для дальнейшей инкапсуляции данных в классе Employee понадобится добавить разнообразные дополнительные методы (такие как GetID(), SetID(), GetCurrentPay(), SetCurrentPay()). В каждом методе, изменяющем данные, может содержаться несколько строк кода, в которых реализована проверка дополнительных бизнес-правил. Несмотря на то что это определенно достижимо, для инкапсуляции данных класса в языке C# имеется удобная альтернативная система записи.

Инкапсуляция с использованием свойств

Хотя инкапсулировать поля данных можно с применением традиционной пары методов get и set, в языках .NET Core предпочтение отдается обеспечению инкапсуляции данных с использованием *свойств*. Прежде всего, имейте в виду, что свойства — всего лишь контейнер для "настоящих" методов доступа и изменения, именуемых get и set соответственно. Следовательно, проектировщик класса по-прежнему может выполнить любую внутреннюю логику перед присваиванием значения (например, преобразовать в верхний регистр, избавиться от недопустимых символов, проверить вхождение внутрь границ и т.д.).

Ниже приведен измененный код класса Employee, который теперь обеспечивает инкапсуляцию каждого поля с использованием синтаксиса свойств вместо традиционных методов get и set.

```
class Employee
{
  // Поля данных.
  private string _empName;
  private int _empId;
  private float _currPay;

  // Свойства!
  public string Name
  {
    get { return _empName; }
    set
    {
      if (value.Length > 15)
      {
        Console.WriteLine("Error! Name length exceeds 15 characters!");
                          // Ошибка! Длина имени превышает 15 символов!
      }
```

```
    else
    {
      _empName = value;
    }
  }
}
// Можно было бы добавить дополнительные бизнес-правила для установки
// данных свойств, но в настоящем примере в этом нет необходимости.
public int Id
{
  get { return _empId; }
  set { _empId = value; }
}
public float Pay
{
  get { return _currPay; }
  set { _currPay = value; }
}
...
}
```

Свойство C# состоит из определений областей `get` (метод доступа) и `set` (метод изменения) прямо внутри самого свойства. Обратите внимание, что свойство указывает тип инкапсулируемых им данных способом, который выглядит как возвращаемое значение. Кроме того, в отличие от метода при определении свойства не применяются круглые скобки (даже пустые). Взгляните на следующий комментарий к текущему свойству `Id`:

```
// int представляет тип данных, инкапсулируемых этим свойством.
public int Id  // Обратите внимание на отсутствие круглых скобок.
{
  get { return _empId; }
  set { _empID = value; }
}
```

В области видимости `set` свойства используется лексема `value`, которая представляет входное значение, присваиваемое свойству вызывающим кодом. Лексема `value` *не* является настоящим ключевым словом C#, а представляет собой то, что называется *контекстным ключевым словом*. Когда лексема `value` находится внутри области `set`, она всегда обозначает значение, присваиваемое вызывающим кодом, и всегда имеет тип, совпадающий с типом самого свойства. Таким образом, вот как свойство `Name` может проверить допустимую длину строки:

```
public string Name
{
  get { return _empName; }
  set
  {
    // Здесь value на самом деле имеет тип string.
    if (value.Length > 15)
    {
      Console.WriteLine("Error! Name length exceeds 15 characters!");
                        // Ошибка! Длина имени превышает 15 символов!
    }
```

```
    else
    {
      empName = value;
    }
  }
}
```

После определения свойств подобного рода вызывающему коду кажется, что он имеет дело с *открытым элементом данных*; однако "за кулисами" при каждом обращении к ним вызывается корректный блок `get` или `set`, предохраняя инкапсуляцию:

```
Console.WriteLine("***** Fun with Encapsulation *****\n");
Employee emp = new Employee("Marvin", 456, 30000);
emp.GiveBonus(1000);
emp.DisplayStats();

// Переустановка и затем получение свойства Name.
emp.Name = "Marv";
Console.WriteLine("Employee is named: {0}", emp.Name);
                // имя сотрудника
Console.ReadLine();
```

Свойства (как противоположность методам доступа и изменения) также облегчают манипулирование типами, поскольку способны реагировать на внутренние операции C#. В целях иллюстрации будем считать, что тип класса `Employee` имеет внутреннюю закрытую переменную-член, представляющую возраст сотрудника. Ниже показаны необходимые изменения (обратите внимание на применение цепочки вызовов конструкторов):

```
class Employee
{
  ...
  // Новое поле и свойство.
  private int _empAge;
  public int Age
  {
    get { return _empAge; }
    set { _empAge = value; }
  }

  // Обновленные конструкторы.
  public Employee() {}
  public Employee(string name, int id, float pay)
    :this(name, 0, id, pay){}
  public Employee(string name, int age, int id, float pay)
  {
    _empName = name;
    _empId = id;
    _empAge = age;
    _currPay = pay;
  }

  // Обновленный метод DisplayStats() теперь учитывает возраст.
  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", _empName);  // имя сотрудника
```

```
    Console.WriteLine("ID: {0}", _empId);
                  // идентификационный номер сотрудника
    Console.WriteLine("Age: {0}", _empAge);    // возраст сотрудника
    Console.WriteLine("Pay: {0}", _currPay);   // текущая выплата
  }
}
```

Теперь предположим, что создан объект `Employee` по имени `joe`. Необходимо сделать так, чтобы в день рождения сотрудника возраст увеличивался на 1 год. Используя традиционные методы `set` и `get`, пришлось бы написать приблизительно такой код:

```
Employee joe = new Employee();
joe.SetAge(joe.GetAge() + 1);
```

Тем не менее, если `empAge` инкапсулируется посредством свойства по имени `Age`, то код будет проще:

```
Employee joe = new Employee();
joe.Age++;
```

Свойства как члены, сжатые до выражений (нововведение в версии 7.0)

Как упоминалось ранее, методы `set` и `get` свойств также могут записываться в виде членов, сжатых до выражений. Правила и синтаксис те же: однострочные методы могут быть записаны с применением нового синтаксиса. Таким образом, свойство `Age` можно было бы переписать следующим образом:

```
public int Age
{
  get => empAge;
  set => empAge = value;
}
```

Оба варианта кода компилируются в одинаковый набор инструкций IL, поэтому выбор используемого синтаксиса зависит только от ваших предпочтений. В книге будут сочетаться оба стиля, чтобы подчеркнуть, что мы не придерживаемся какого-то специфического стиля написания кода.

Использование свойств внутри определения класса

Свойства, в частности их порция `set`, являются общепринятым местом для размещения бизнес-правил класса. В текущий момент класс `Employee` имеет свойство `Name`, которое гарантирует, что длина имени не превышает 15 символов. Остальные свойства (`ID`, `Pay` и `Age`) также могут быть обновлены соответствующей логикой.

Хотя все это хорошо, но необходимо также принимать во внимание и то, что обычно происходит внутри конструктора класса. Конструктор получает входные параметры, проверяет данные на предмет допустимости и затем присваивает значения внутренним закрытым полям. Пока что главный конструктор *не* проверяет входные строковые данные на вхождение в диапазон допустимых значений, а потому его можно было бы изменить следующим образом:

```
public Employee(string name, int age, int id, float pay)
{
  // Похоже на проблему...
  if (name.Length > 15)
```

```
  {
    Console.WriteLine("Error! Name length exceeds 15 characters!");
                    // Ошибка! Длина имени превышает 15 символов!
  }
  else
  {
    _empName = name;
  }
  _empId = id;
  _empAge = age;
  _currPay = pay;
}
```

Наверняка вы заметили проблему, связанную с таким подходом. Свойство `Name` и главный конструктор выполняют одну и ту же проверку на наличие ошибок. Реализуя проверки для других элементов данных, есть реальный шанс столкнуться с дублированием кода. Стремясь рационализировать код и изолировать всю проверку, касающуюся ошибок, в каком-то центральном местоположении, вы добьетесь успеха, если для получения и установки значений внутри класса *всегда* будете применять свойства. Взгляните на показанный ниже модифицированный конструктор:

```
public Employee(string name, int age, int id, float pay)
{
  // Уже лучше! Используйте свойства для установки данных класса.
  // Это сократит количество дублированных проверок на предмет ошибок.
  Name = name;
  Age = age;
  ID = id;
  Pay = pay;
}
```

Помимо обновления конструкторов для применения свойств при присваивании значений рекомендуется повсюду в реализации класса использовать свойства, чтобы гарантировать неизменное соблюдение бизнес-правил. Во многих случаях прямая ссылка на лежащие в основе закрытые данные производится только внутри самого свойства. Имея все сказанное в виду, модифицируйте класс `Employee`:

```
class Employee
{
  // Поля данных.
  private string _empName;
  private int _empId;
  private float _currPay;
  private int _empAge;

  // Конструкторы.
  public Employee() { }
  public Employee(string name, int id, float pay)
    :this(name, 0, id, pay){}
  public Employee(string name, int age, int id, float pay)
  {
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
  }
```

```
  // Методы.
  public void GiveBonus(float amount) => Pay += amount;
  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", Name);  // имя сотрудника
    Console.WriteLine("ID: {0}", Id);
                    // идентификационный номер сотрудника
    Console.WriteLine("Age: {0}", Age);    // возраст сотрудника
    Console.WriteLine("Pay: {0}", Pay);    // текущая выплата
  }
  // Свойства остаются прежними...
  ...
}
```

Свойства, допускающие только чтение

При инкапсуляции данных может возникнуть желание сконфигурировать *свойство, допускающее только чтение*, для чего нужно просто опустить блок `set`. Например, пусть имеется новое свойство по имени `SocialSecurityNumber`, которое инкапсулирует закрытую строковую переменную `empSSN`. Вот как превратить его в свойство, доступное только для чтения:

```
public string SocialSecurityNumber
{
  get { return _empSSN; }
}
```

Свойства, которые имеют только метод `get`, можно упростить с использованием членов, сжатых до выражений. Следующая строка эквивалентна предыдущему блоку кода:

```
public string SocialSecurityNumber => _empSSN;
```

Теперь предположим, что конструктор класса принимает новый параметр, который дает возможность указывать в вызывающем коде номер карточки социального страхования для объекта, представляющего сотрудника. Поскольку свойство `SocialSecurityNumber` допускает только чтение, устанавливать значение так, как показано ниже, нельзя:

```
public Employee(string name, int age, int id, float pay, string ssn)
{
  Name = name;
  Age = age;
  ID = id;
  Pay = pay;
  // Если свойство предназначено только для чтения, это больше невозможно!
  SocialSecurityNumber = ssn;
}
```

Если только вы не готовы переделать данное свойство в поддерживающее чтение и запись (что вскоре будет сделано), тогда единственным вариантом со свойствами, допускающими только чтение, будет применение лежащей в основе переменной-члена `empSSN` внутри логики конструктора:

```
public Employee(string name, int age, int id, float pay, string ssn)
{
  ...
  // Проверить надлежащим образом входной параметр ssn
  // и затем установить значение.
  empSSN = ssn;
}
```

Свойства, допускающие только запись

Если вы хотите сконфигурировать *свойство как допускающее только запись*, тогда опустите блок `get`, например:

```
public int Id
{
  set { _empId = value; }
}
```

Смешивание закрытых и открытых методов `get`/`set` в свойствах

При определении свойств уровень доступа для методов `get` и `set` может быть разным. Возвращаясь к номеру карточки социального страхования, если цель заключается в том, чтобы предотвратить модификацию номера *извне* класса, тогда объявите метод `get` как открытый, но метод `set` — как закрытый:

```
public string SocialSecurityNumber
{
  get => _empSSN;
  private set => _empSSN = value;
}
```

Обратите внимание, что это превращает свойство, допускающее только чтение, в допускающее чтение и запись. Отличие в том, что запись скрыта от чего-либо за рамками определяющего класса.

Еще раз о ключевом слове `static`: определение статических свойств

Ранее в главе рассказывалось о роли ключевого слова `static`. Теперь, когда вы научились использовать синтаксис свойств C#, мы можем формализовать статические свойства. В проекте `StaticDataAndMembers` класс `SavingsAccount` имел два открытых статических метода для получения и установки процентной ставки. Однако более стандартный подход предусматривает помещение такого элемента данных в статическое свойство. Ниже приведен пример (обратите внимание на применение ключевого слова `static`):

```
// Простой класс депозитного счета.
class SavingsAccount
{
  // Данные уровня экземпляра.
  public double currBalance;

  // Статический элемент данных.
  private static double _currInterestRate = 0.04;
```

```
  // Статическое свойство.
  public static double InterestRate
  {
    get { return _currInterestRate; }
    set { _currInterestRate = value; }
  }
  ...
}
```

Если вы хотите использовать свойство `InterestRate` вместо предыдущих статических методов, тогда можете модифицировать свой код следующим образом:

```
// Вывести текущую процентную ставку через свойство.
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.InterestRate);
```

Сопоставление с образцом и шаблоны свойств (нововведение в версии 8.0)

Шаблон свойств позволяет сопоставлять со свойствами объекта. В качестве примера добавьте к проекту новый файл (`EmployeePayTypeEnum.cs`) и определите в нем перечисление для типов оплаты сотрудников:

```
namespace EmployeeApp
{
  public enum EmployeePayTypeEnum
  {
    Hourly,         // почасовая оплата
    Salaried,       // оклад
    Commission      // комиссионное вознаграждение
  }
}
```

Обновите класс `Employee`, добавив свойство для типа оплаты и инициализировав его в конструкторе. Ниже показаны изменения, которые понадобится внести в код:

```
private EmployeePayTypeEnum _payType;
public EmployeePayTypeEnum PayType
{
  get => _payType;
  set => _payType = value;
}
public Employee(string name, int id, float pay, string empSsn)
  : this(name,0,id,pay, empSsn, EmployeePayTypeEnum.Salaried)
{
}
public Employee(string name, int age, int id,
  float pay, string empSsn, EmployeePayTypeEnum payType)
{
  Name = name;
  Id = id;
  Age = age;
  Pay = pay;
  SocialSecurityNumber = empSsn;
  PayType = payType;
}
```

Теперь, когда все элементы на месте, метод `GiveBonus()` можно обновить на основе типа оплаты сотрудника. Сотрудники с комиссионным вознаграждением получают премию 10%, с почасовой оплатой — 40-часовый эквивалент соответствующей премии, а с окладом — введенную сумму. Вот модифицированный код метода `GiveBonus()`:

```
public void GiveBonus(float amount)
{
  Pay = this switch
  {
    {PayType: EmployeePayTypeEnum.Commission }
      => Pay += .10F * amount,
    {PayType: EmployeePayTypeEnum.Hourly }
      => Pay += 40F * amount/2080F,
    {PayType: EmployeePayTypeEnum.Salaried }
      => Pay += amount,
    _ => Pay+=0
  };
}
```

Как и с другими операторами `switch`, в которых используется сопоставление с образцом, должен быть предусмотрен общий оператор `case` или же оператор `switch` обязан генерировать исключение, если ни один из операторов `case` не был удовлетворен.

Чтобы протестировать внесенные обновления, добавьте к операторам верхнего уровня следующий код:

```
Employee emp = new Employee("Marvin", 45, 123, 1000, "111-11-1111",
  EmployeePayTypeEnum.Salaried);
Console.WriteLine(emp.Pay);
emp.GiveBonus(100);
Console.WriteLine(emp.Pay);
```

Понятие автоматических свойств

При создании свойств для инкапсуляции данных часто обнаруживается, что области `set` содержат код для применения бизнес-правил программы. Тем не менее, в некоторых случаях нужна только простая логика извлечения или установки значения. В результате получается большой объем кода следующего вида:

```
// Тип Car, использующий стандартный синтаксис свойств.
class Car
{
  private string carName = "";
  public string PetName
  {
    get { return carName; }
    set { carName = value; }
  }
}
```

В подобных случаях многократное определение закрытых поддерживающих полей и простых свойств может стать слишком громоздким. Например, при построении

класса, которому нужны девять закрытых элементов данных, в итоге получаются девять связанных с ними свойств, которые представляют собой не более чем тонкие оболочки для служб инкапсуляции.

Чтобы упростить процесс обеспечения простой инкапсуляции данных полей, можно использовать *синтаксис автоматических свойств*. Как следует из названия, это средство перекладывает работу по определению закрытых поддерживающих полей и связанных с ними свойств C# на компилятор за счет применения небольшого нововведения в синтаксисе. В целях иллюстрации создайте новый проект консольного приложения по имени `AutoProps` и добавьте к нему файл `Car.cs` с переделанным классом `Car`, в котором данный синтаксис используется для быстрого создания трех свойств:

```
using System;

namespace AutoProps
{
  class Car
  {
    // Автоматические свойства! Нет нужды определять поддерживающие поля.
    public string PetName { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }
  }
}
```

На заметку! Среды Visual Studio и Visual Studio Code предоставляют фрагмент кода `prop`. Если вы наберете слово `prop` внутри определения класса и нажмете клавишу <Tab>, то IDE-среда сгенерирует начальный код для нового автоматического свойства. Затем с помощью клавиши <Tab> можно циклически проходить по всем частям определения и заполнять необходимые детали. Испытайте описанный прием.

При определении автоматического свойства вы просто указываете модификатор доступа, лежащий в основе тип данных, имя свойства и пустые области `get/set`. Во время компиляции тип будет оснащен автоматически сгенерированным поддерживающим полем и подходящей реализацией логики `get/set`.

На заметку! Имя автоматически сгенерированного закрытого поддерживающего поля будет невидимым для вашей кодовой базы C#. Просмотреть его можно только с помощью инструмента вроде `ildasm.exe`.

Начиная с версии C# 6, разрешено определять "автоматическое свойство только для чтения", опуская область `set`. Автоматические свойства только для чтения можно устанавливать только в конструкторе. Тем не менее, определять свойство, предназначенное только для записи, нельзя. Вот пример:

```
// Свойство только для чтения? Допустимо!
public int MyReadOnlyProp { get; }

// Свойство только для записи? Ошибка!
public int MyWriteOnlyProp { set; }
```

Взаимодействие с автоматическими свойствами

Поскольку компилятор будет определять закрытые поддерживающие поля на этапе компиляции (и учитывая, что эти поля в коде C# непосредственно не доступны), в классе, который имеет автоматические свойства, для установки и чтения лежащих в их основе значений всегда должен применяться синтаксис свойств. Указанный факт важно отметить, т.к. многие программисты напрямую используют закрытые поля *внутри* определения класса, что в данном случае невозможно. Например, если бы класс `Car` содержал метод `DisplayStats()`, то в его реализации пришлось бы применять имена свойств:

```
class Car
{
  // Автоматические свойства!
  public string PetName { get; set; }
  public int Speed { get; set; }
  public string Color { get; set; }

  public void DisplayStats()
  {
    Console.WriteLine("Car Name: {0}", PetName);
    Console.WriteLine("Speed: {0}", Speed);
    Console.WriteLine("Color: {0}", Color);
  }
}
```

При использовании экземпляра класса, определенного с автоматическими свойствами, присваивать и получать значения можно с помощью вполне ожидаемого синтаксиса свойств:

```
using System;
using AutoProps;

Console.WriteLine("***** Fun with Automatic Properties *****\n");

Car c = new Car();
c.PetName = "Frank";
c.Speed = 55;
c.Color = "Red";

Console.WriteLine("Your car is named {0}? That's odd...",
  c.PetName);
c.DisplayStats();

Console.ReadLine();
```

Автоматические свойства и стандартные значения

Когда автоматические свойства применяются для инкапсуляции числовых и булевских данных, их можно использовать прямо внутри кодовой базы, т.к. скрытым поддерживающим полям будут присваиваться безопасные стандартные значения (`false` для булевских и `0` для числовых данных). Но имейте в виду, что когда синтаксис автоматического свойства применяется для упаковки переменной другого класса, то скрытое поле ссылочного типа также будет установлено в стандартное значение `null` (и это может привести к проблеме, если не проявить должную осторожность).

Добавьте к текущему проекту новый файл класса по имени `Garage` (представляющий гараж), в котором используются два автоматических свойства (разумеется, ре-

альный класс гаража может поддерживать коллекцию объектов Car; однако в данный момент проигнорируем такую деталь):

```
namespace AutoProps
{
  class Garage
  {
    // Скрытое поддерживающее поле int установлено в 0!
    public int NumberOfCars { get; set; }

    // Скрытое поддерживающее поле Car установлено в null!
    public Car MyAuto { get; set; }
  }
}
```

Имея стандартные значения C# для полей данных, значение NumberOfCars можно вывести в том виде, как есть (поскольку ему автоматически присвоено значение 0). Но если напрямую обратиться к MyAuto, то во время выполнения сгенерируется исключение ссылки на null, потому что лежащей в основе переменной-члену типа Car не был присвоен новый объект.

```
...
Garage g = new Garage();

// Нормально, выводится стандартное значение 0.
Console.WriteLine("Number of Cars: {0}", g.NumberOfCars);

// Ошибка во время выполнения! Поддерживающее поле в данный момент
равно null!
Console.WriteLine(g.MyAuto.PetName);
Console.ReadLine();
```

Чтобы решить проблему, можно модифицировать конструкторы класса, обеспечив безопасное создание объекта. Ниже показан пример:

```
class Garage
{
  // Скрытое поддерживающее поле установлено в 0!
  public int NumberOfCars { get; set; }

  // Скрытое поддерживающее поле установлено в null!
  public Car MyAuto { get; set; }

  // Для переопределения стандартных значений, присвоенных скрытым
  // поддерживающим полям, должны использоваться конструкторы.
  public Garage()
  {
    MyAuto = new Car();
    NumberOfCars = 1;
  }
  public Garage(Car car, int number)
  {
    MyAuto = car;
    NumberOfCars = number;
  }
}
```

После такого изменения объект Car теперь можно помещать в объект Garage:

```
Console.WriteLine("***** Fun with Automatic Properties *****\n");

// Создать объект автомобиля.
Car c = new Car();
c.PetName = "Frank";
c.Speed = 55;
c.Color = "Red";
c.DisplayStats();

// Поместить автомобиль в гараж.
Garage g = new Garage();
g.MyAuto = c;
// Вывести количество автомобилей в гараже.
Console.WriteLine("Number of Cars in garage: {0}", g.NumberOfCars);
// Вывести название автомобиля.
Console.WriteLine("Your car is named: {0}", g.MyAuto.PetName);

Console.ReadLine();
```

Инициализация автоматических свойств

Наряду с тем, что предыдущий подход работает вполне нормально, в версии C# 6 появилась языковая возможность, которая содействует упрощению способа присваивания автоматическим свойствам их начальных значений. Как упоминалось ранее в главе, полю данных в классе можно напрямую присваивать начальное значение при его объявлении. Например:

```
class Car
{
  private int numberOfDoors = 2;
}
```

В похожей манере язык C# теперь позволяет присваивать начальные значения лежащим в основе поддерживающим полям, которые генерируются компилятором. В результате смягчаются трудности, присущие добавлению операторов кода в конструкторы класса, которые обеспечивают корректную установку данных свойств.

Ниже приведена модифицированная версия класса `Garage` с инициализацией автоматических свойств подходящими значениями. Обратите внимание, что больше нет необходимости в добавлении к стандартному конструктору класса логики для выполнения безопасного присваивания. В коде свойству `MyAuto` напрямую присваивается новый объект `Car`.

```
class Garage
{
  // Скрытое поддерживающее поле установлено в 1.
  public int NumberOfCars { get; set; } = 1;

  // Скрытое поддерживающее поле установлено в новый объект Car.
  public Car MyAuto { get; set; } = new Car();

  public Garage(){}
  public Garage(Car car, int number)
  {
    MyAuto = car;
    NumberOfCars = number;
  }
}
```

Наверняка вы согласитесь с тем, что автоматические свойства — очень полезное средство языка программирования C#, т.к. отдельные свойства в классе можно определять с применением модернизированного синтаксиса. Конечно, если вы создаете свойство, которое помимо получения и установки закрытого поддерживающего поля требует дополнительного кода (такого как логика проверки достоверности, регистрация в журнале событий, взаимодействие с базой данных и т.д.), то его придется определять как "нормальное" свойство .NET Core вручную. Автоматические свойства C# не делают ничего кроме обеспечения простой инкапсуляции для лежащей в основе порции (сгенерированных компилятором) закрытых данных.

Понятие инициализации объектов

На протяжении всей главы можно заметить, что при создании нового объекта конструктор позволяет указывать начальные значения. Вдобавок свойства позволяют безопасным образом получать и устанавливать лежащие в основе данные. При работе со сторонними классами, включая классы из библиотеки базовых классов .NET Core, нередко обнаруживается, что в них отсутствует конструктор, который позволял бы устанавливать абсолютно все порции данных состояния. В итоге программист обычно вынужден выбирать наилучший конструктор из числа возможных и затем присваивать остальные значения с использованием предоставляемого набора свойств.

Обзор синтаксиса инициализации объектов

Для упрощения процесса создания и подготовки объекта в C# предлагается *синтаксис инициализации объектов*. Такой прием делает возможным создание новой объектной переменной и присваивание значений многочисленным свойствам и/или открытым полям в нескольких строках кода. Синтаксически инициализатор объекта выглядит как список разделенных запятыми значений, помещенный в фигурные скобки ({}). Каждый элемент в списке инициализации отображается на имя открытого поля или открытого свойства инициализируемого объекта.

Чтобы увидеть данный синтаксис в действии, создайте новый проект консольного приложения по имени `ObjectInitializers`. Ниже показан класс `Point`, в котором присутствуют автоматические свойства (для синтаксиса инициализации объектов они не обязательны, но помогают получить более лаконичный код):

```
class Point
{
  public int X { get; set; }
  public int Y { get; set; }

  public Point(int xVal, int yVal)
  {
    X = xVal;
    Y = yVal;
  }
  public Point() { }

  public void DisplayStats()
  {
    Console.WriteLine("[{0}, {1}]", X, Y);
  }
}
```

А теперь посмотрим, как создавать объекты Point, с применением любого из следующих подходов:

```
Console.WriteLine("***** Fun with Object Init Syntax *****\n");

// Создать объект Point, устанавливая каждое свойство вручную.
Point firstPoint = new Point();
firstPoint.X = 10;
firstPoint.Y = 10;
firstPoint.DisplayStats();

// Или создать объект Point посредством специального конструктора.
Point anotherPoint = new Point(20, 20);
anotherPoint.DisplayStats();

// Или создать объект Point, используя синтаксис инициализации объектов.
Point finalPoint = new Point { X = 30, Y = 30 };
finalPoint.DisplayStats();
Console.ReadLine();
```

При создании последней переменной Point специальный конструктор не используется (как делается традиционно), а взамен устанавливаются значения открытых свойств X и Y. "За кулисами" вызывается стандартный конструктор типа, за которым следует установка значений указанных свойств. В таком отношении синтаксис инициализации объектов представляет собой просто сокращение синтаксиса для создания переменной класса с применением стандартного конструктора и установки данных состояния свойство за свойством.

На заметку! Важно помнить о том, что процесс инициализации объектов неявно использует методы установки свойств. Если метод установки какого-то свойства помечен как private, тогда этот синтаксис применить не удастся.

Использование средства доступа только для инициализации (нововведение в версии 9.0)

В версии C# 9.0 появилось новое средство доступа только для инициализации. Оно позволяет устанавливать свойство во время инициализации, но после завершения конструирования объекта свойство становится доступным только для чтения. Свойства такого типа называются *неизменяемыми*. Добавьте к проекту новый файл класса по имени ReadOnlyPointAfterCreation.cs и поместите в него следующий код:

```
using System;

namespace ObjectInitializers
{
  class PointReadOnlyAfterCreation
  {
    public int X { get; init; }
    public int Y { get; init; }

    public void DisplayStats()
    {
      Console.WriteLine("InitOnlySetter: [{0}, {1}]", X, Y);
    }
```

```
    public PointReadOnlyAfterCreation(int xVal, int yVal)
    {
      X = xVal;
      Y = yVal;
    }
    public PointReadOnlyAfterCreation() { }
  }
}
```

Новый класс тестируется с применением приведенного ниже кода:

```
// Создать объект точки, допускающий только чтение
// после конструирования.
PointReadOnlyAfterCreation firstReadonlyPoint =
   new PointReadOnlyAfterCreation(20, 20);
firstReadonlyPoint.DisplayStats();

// Или создать объект точки с использованием синтаксиса только
// для инициализации.
PointReadOnlyAfterCreation secondReadonlyPoint =
   new PointReadOnlyAfterCreation
  { X = 30, Y = 30 };
secondReadonlyPoint.DisplayStats();
```

Обратите внимание, что в коде для класса `Point` ничего не изменилось кроме, разумеется, имени класса. Отличие в том, что после создания экземпляра класса модифицировать значения свойств `X` и `Y` нельзя. Например, показанный далее код не скомпилируется:

```
// Следующие две строки не скомпилируются.
secondReadonlyPoint.X = 10;
secondReadonlyPoint.Y = 10;
```

Вызов специальных конструкторов с помощью синтаксиса инициализации

В предшествующих примерах объекты типа `Point` инициализировались путем неявного вызова стандартного конструктора этого типа:

```
// Здесь стандартный конструктор вызывается неявно.
Point finalPoint = new Point { X = 30, Y = 30 };
```

При желании стандартный конструктор допускается вызывать и явно:

```
// Здесь стандартный конструктор вызывается явно.
Point finalPoint = new Point() { X = 30, Y = 30 };
```

Имейте в виду, что при конструировании объекта типа с использованием синтаксиса инициализации можно вызывать *любой* конструктор, определенный в классе. В настоящий момент в типе `Point` определен конструктор с двумя аргументами для установки позиции (*x*, *y*). Таким образом, следующее объявление переменной `Point` приведет к установке `X` в `100` и `Y` в `100` независимо от того факта, что в аргументах конструктора указаны значения `10` и `16`:

```
// Вызов специального конструктора.
Point pt = new Point(10, 16) { X = 100, Y = 100 };
```

Имея текущее определение типа Point, вызов специального конструктора с применением синтаксиса инициализации не особенно полезен (и излишне многословен). Тем не менее, если тип Point предоставляет новый конструктор, который позволяет вызывающему коду устанавливать цвет (через специальное перечисление PointColor), тогда комбинация специальных конструкторов и синтаксиса инициализации объектов становится ясной.

Добавьте к проекту новый файл класса по имени PointColorEnum.cs и создайте следующее перечисление цветов:

```
namespace ObjectInitializers
{
  enum PointColorEnum
  {
    LightBlue,
    BloodRed,
    Gold
  }
}
```

Обновите код класса Point, как показано ниже:

```
class Point
{
  public int X { get; set; }
  public int Y { get; set; }
  public PointColorEnum Color{ get; set; }

  public Point(int xVal, int yVal)
  {
    X = xVal;
    Y = yVal;
    Color = PointColorEnum.Gold;
  }

  public Point(PointColorEnum ptColor)
  {
    Color = ptColor;
  }

  public Point() : this(PointColorEnum.BloodRed){ }

  public void DisplayStats()
  {
    Console.WriteLine("[{0}, {1}]", X, Y);
    Console.WriteLine("Point is {0}", Color);
  }
}
```

Посредством нового конструктора теперь можно создавать точку золотистого цвета (в позиции (90, 20)):

```
// Вызов более интересного специального конструктора
// с помощью синтаксиса инициализации.
Point goldPoint = new Point(PointColorEnum.Gold){ X = 90, Y = 20 };
goldPoint.DisplayStats();
```

Инициализация данных с помощью синтаксиса инициализации

Как кратко упоминалось ранее в главе (и будет подробно обсуждаться в главе 6), отношение "имеет" позволяет формировать новые классы, определяя переменные-члены существующих классов. Например, пусть определен класс `Rectangle`, в котором для представления координат верхнего левого и нижнего правого углов используется тип `Point`. Так как автоматические свойства устанавливают все переменные с типами классов в `null`, новый класс будет реализован с применением "традиционного" синтаксиса свойств:

```
using System;
namespace ObjectInitializers
{
  class Rectangle
  {
    private Point topLeft = new Point();
    private Point bottomRight = new Point();

    public Point TopLeft
    {
      get { return topLeft; }
      set { topLeft = value; }
    }
    public Point BottomRight
    {
      get { return bottomRight; }
      set { bottomRight = value; }
    }

    public void DisplayStats()
    {
      Console.WriteLine("[TopLeft: {0}, {1}, {2} BottomRight: {3},
{4}, {5}]",
        topLeft.X, topLeft.Y, topLeft.Color,
        bottomRight.X, bottomRight.Y, bottomRight.Color);
    }
  }
}
```

С помощью синтаксиса инициализации объектов можно было бы создать новую переменную `Rectangle` и установить внутренние объекты `Point` следующим образом:

```
// Создать и инициализировать объект Rectangle.
Rectangle myRect = new Rectangle
{
   TopLeft = new Point { X = 10, Y = 10 },
   BottomRight = new Point { X = 200, Y = 200}
};
```

Преимущество синтаксиса инициализации объектов в том, что он по существу сокращает объем вводимого кода (предполагая отсутствие подходящего конструктора). Вот как выглядит традиционный подход к созданию похожего экземпляра `Rectangle`:

```
// Традиционный подход.
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point();
p2.X = 200;
p2.Y = 200;
r.BottomRight = p2;
```

Поначалу синтаксис инициализации объектов может показаться несколько непривычным, но как только вы освоитесь с кодом, то будете приятно поражены тем, насколько быстро и с минимальными усилиями можно устанавливать состояние нового объекта.

Работа с константными полями данных и полями данных, допускающими только чтение

Иногда требуется свойство, которое вы вообще не хотите изменять либо с момента компиляции, либо с момента его установки во время конструирования, также известное как *неизменяемое*. Один пример уже был исследован ранее — средства доступа только для инициализации. А теперь мы займемся константными полями и полями, допускающими только чтение.

Понятие константных полей данных

Язык C# предлагает ключевое слово `const`, предназначенное для определения константных данных, которые после начальной установки больше никогда не могут быть изменены. Как нетрудно догадаться, оно полезно при определении набора известных значений для использования в приложениях, логически связанных с заданным классом или структурой.

Предположим, что вы строите обслуживающий класс по имени `MyMathClass`, в котором нужно определить значение числа π (для простоты будем считать его равным `3.14`). Начните с создания нового проекта консольного приложения по имени `ConstData` и добавьте к нему файл класса `MyMathClass.cs`. Учитывая, что давать возможность другим разработчикам изменять это значение в коде нежелательно, число π можно смоделировать с помощью следующей константы:

```
// MyMathClass.cs
using System;
namespace ConstData
{
  class MyMathClass
  {
    public const double PI = 3.14;
  }
}
```

Приведите код в файле `Program.cs` к следующему виду:

```
using System;
using ConstData;
```

```
Console.WriteLine("***** Fun with Const *****\n");
Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
// Ошибка! Константу изменять нельзя!
// MyMathClass.PI = 3.1444;
Console.ReadLine();
```

Обратите внимание, что ссылка на константные данные, определенные в классе `MyMathClass`, производится с применением префикса в виде имени класса (т.е. `MyMathClass.PI`). Причина в том, что константные поля класса являются неявно *статическими*. Однако допустимо определять локальные константные данные и обращаться к ним внутри области действия метода или свойства, например:

```
static void LocalConstStringVariable()
{
  // Доступ к локальным константным данным можно получать напрямую.
  const string fixedStr = "Fixed string Data";
  Console.WriteLine(fixedStr);
  // Ошибка!
  // fixedStr = "This will not work!";
}
```

Независимо от того, где вы определяете константную часть данных, всегда помните о том, что начальное значение, присваиваемое константе, должно быть указано в момент ее определения. Присваивание значения `PI` внутри конструктора класса приводит к ошибке на этапе компиляции:

```
class MyMathClass
{
  // Попытка установить PI в конструкторе?
  public const double PI;
  public MyMathClass()
  {
    // Невозможно - присваивание должно осуществляться в момент объявления.
    PI = 3.14;
  }
}
```

Такое ограничение связано с тем фактом, что значение константных данных должно быть известно на этапе компиляции. Как известно, конструкторы (или любые другие методы) вызываются во время выполнения.

Понятие полей данных, допускающих только чтение

С константными данными тесно связано понятие *полей данных, допускающих только чтение* (которое не следует путать со свойствами, доступными только для чтения). Подобно константе поле только для чтения нельзя изменять после первоначального присваивания, иначе вы получите ошибку на этапе компиляции. Тем не менее, в отличие от константы значение, присваиваемое такому полю, может быть определено во время выполнения и потому может на законном основании присваиваться внутри конструктора, но больше нигде.

Поле только для чтения полезно в ситуации, когда значение не известно вплоть до стадии выполнения (возможно из-за того, что для его получения необходимо прочитать внешний файл), но нужно гарантировать, что впоследствии оно не будет изменяться. В целях иллюстрации рассмотрим следующую модификацию класса `MyMathClass`:

```
class MyMathClass
{
  // Поля только для чтения могут присваиваться
  // в конструкторах, но больше нигде.
  public readonly double PI;

  public MyMathClass ()
  {
    PI = 3.14;
  }
}
```

Любая попытка выполнить присваивание полю, помеченному как `readonly`, за пределами конструктора приведет к ошибке на этапе компиляции:

```
class MyMathClass
{
  public readonly double PI;
  public MyMathClass ()
  {
    PI = 3.14;
  }
  // Ошибка!
  public void ChangePI()
  { PI = 3.14444;}
}
```

Понятие статических полей, допускающих только чтение

В отличие от константных полей поля, допускающие только чтение, не являются неявно статическими. Таким образом, если необходимо предоставить доступ к `PI` на уровне класса, то придется явно использовать ключевое слово `static`. Если значение статического поля только для чтения известно на этапе компиляции, тогда начальное присваивание выглядит очень похожим на такое присваивание в случае константы (однако в этой ситуации проще применить ключевое слово `const`, потому что поле данных присваивается в момент его объявления):

```
class MyMathClass
{
  public static readonly double PI = 3.14;
}
// Program.cs
Console.WriteLine("***** Fun with Const *****");
Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
Console.ReadLine();
```

Тем не менее, если значение статического поля только для чтения не известно вплоть до времени выполнения, то должен использоваться статический конструктор, как было описано ранее в главе:

```
class MyMathClass
{
  public static readonly double PI;
  static MyMathClass()
  { PI = 3.14; }
}
```

Понятие частичных классов

При работе с классами важно понимать роль ключевого слова `partial` языка C#. Ключевое слово `partial` позволяет разбить одиночный класс на множество файлов кода. Когда вы создаете шаблонные классы Entity Framework Core из базы данных, то все полученные в результате классы будут частичными. Таким образом, любой код, который вы написали для дополнения этих файлов, не будет перезаписан при условии, что код находится в отдельных файлах классов, помеченных с помощью ключевого слова `partial`. Еще одна причина связана с тем, что ваш класс может со временем разрастись и стать трудным в управлении, и в качестве промежуточного шага к его рефакторингу вы разбиваете код на части.

В языке C# одиночный класс можно разносить по нескольким файлам кода для отделения стереотипного кода от более полезных (и сложных) членов. Чтобы ознакомиться с ситуацией, когда частичные классы могут быть удобными, загрузите ранее созданный проект `EmployeeApp` в Visual Studio и откройте файл `Employee.cs` для редактирования. Как вы помните, этот единственный файл содержит код для всех аспектов класса:

```
class Employee
{
  // Поля данных
  // Конструкторы
  // Методы
  // Свойства
}
```

С применением частичных классов вы могли бы перенести (скажем) свойства, конструкторы и поля данных в новый файл по имени `Employee.Core.cs` (имя файла к делу не относится). Первый шаг предусматривает добавление ключевого слова `partial` к текущему определению класса и вырезание кода, подлежащего помещению в новый файл:

```
// Employee.cs
partial class Employee
{
  // Методы
  // Свойства
}
```

Далее предположив, что к проекту был добавлен новый файл класса, в него можно переместить поля данных и конструкторы с помощью простой операции вырезания и вставки. Кроме того, вы *обязаны* добавить ключевое слово `partial` к этому аспекту определения класса. Вот пример:

```
// Employee.Core.cs
partial class Employee
{
  // Поля данных
  // Конструкторы
}
```

На заметку! Не забывайте, что каждый частичный класс должен быть помечен ключевым словом `partial`!

После компиляции модифицированного проекта вы не должны заметить вообще никакой разницы. Вся идея, положенная в основу частичного класса, касается только стадии проектирования. Как только приложение скомпилировано, в сборке оказывается один целостный класс. Единственное требование при определении частичных классов связано с тем, что разные части должны иметь одно и то же имя класса и находиться внутри того же самого пространства имен .NET Core.

Использование записей (нововведение в версии 9.0)

В версии C# 9.0 появился особый вид классов — *записи*. Записи являются ссылочными типами, которые предоставляют синтезированные методы с целью обеспечения семантики значений для эквивалентности. По умолчанию типы записей неизменяемы. Хотя по существу дела вы могли бы создать неизменяемый класс, но с применением комбинации средств доступа только для инициализации и свойств, допускающих только чтение, типы записей позволяют избавиться от такой дополнительной работы.

Чтобы приступить к экспериментам с записями, создайте новый проект консольного приложения по имени `FunWithRecords`. Измените код класса `Car` из примеров, приведенных ранее в главе:

```
class Car
{
  public string Make { get; set; }
  public string Model { get; set; }
  public string Color { get; set; }

  public Car() {}

  public Car(string make, string model, string color)
  {
    Make = make;
    Model = model;
    Color = color;
  }
}
```

Как вы уже хорошо знаете, после создания экземпляра этого класса вы можете изменять любое свойство во время выполнения. Если каждый экземпляр должен быть неизменяемым, тогда можете модифицировать определения свойств следующим образом:

```
public string Make { get; init; }
public string Model { get; init; }
public string Color { get; init; }
```

Для использования нового класса `Car` в показанном ниже коде из файла `Program.cs` создаются два его экземпляра — один через инициализацию объекта, а другой посредством специального конструктора:

```
using System;
using FunWithRecords;

Console.WriteLine("Fun with Records!");

// Использовать инициализацию объекта.
Car myCar = new Car
{
  Make = "Honda",
  Model = "Pilot",
  Color = "Blue"
};
Console.WriteLine("My car: ");
DisplayCarStats(myCar);
Console.WriteLine();

// Использовать специальный конструктор.
Car anotherMyCar = new Car("Honda", "Pilot", "Blue");
Console.WriteLine("Another variable for my car: ");
DisplayCarStats(anotherMyCar);
Console.WriteLine();

// Попытка изменения свойства приводит к ошибке на этапе компиляции.
// myCar.Color = "Red";

Console.ReadLine();

static void DisplayCarStats(Car c)
{
  Console.WriteLine("Car Make: {0}", c.Make);
  Console.WriteLine("Car Model: {0}", c.Model);
  Console.WriteLine("Car Color: {0}", c.Color);
}
```

Вполне ожидаемо оба метода создания объекта работают, значения свойств отображаются, а попытка изменить свойство после конструирования приводит к ошибке на этапе компиляции.

Чтобы создать тип записи `CarRecord`, добавьте к проекту новый файл по имени `CarRecord.cs` со следующим кодом:

```
record CarRecord
{
  public string Make { get; init; }
  public string Model { get; init; }
  public string Color { get; init; }

  public CarRecord () {}

  public CarRecord (string make, string model, string color)
  {
    Make = make;
    Model = model;
    Color = color;
  }
}
```

Запустив приведенный далее код из `Program.cs`, вы можете удостовериться в том, что поведение записи `CarRecord` будет таким же, как у класса `Car` со средствами доступа только для инициализации:

```
Console.WriteLine("/*************** RECORDS ********************/");
// Использовать инициализацию объекта.
CarRecord myCarRecord = new CarRecord
{
  Make = "Honda",
  Model = "Pilot",
  Color = "Blue"
};
Console.WriteLine("My car: ");
DisplayCarRecordStats(myCarRecord);
Console.WriteLine();

// Использовать специальный конструктор.
CarRecord anotherMyCarRecord = new CarRecord("Honda", "Pilot",
"Blue");
Console.WriteLine("Another variable for my car: ");
Console.WriteLine(anotherMyCarRecord.ToString());
Console.WriteLine();

// Попытка изменения свойства приводит к ошибке на этапе компиляции.
// myCarRecord.Color = "Red";

Console.ReadLine();
```

Хотя мы пока еще не обсуждали эквивалентность (см. следующий раздел) или наследование (см. следующую главу) с типами записей, первое знакомство с записями не создает впечатления, что они обеспечивают большое преимущество. Текущий пример записи `CarRecord` включал весь ожидаемый связующий код. Заметное отличие присутствует в выводе: метод `ToString()` для типов записей более причудлив, как видно в показанном ниже фрагменте вывода:

```
/*************** RECORDS ********************/
My car:
CarRecord { Make = Honda, Model = Pilot, Color = Blue }
Another variable for my car:
CarRecord { Make = Honda, Model = Pilot, Color = Blue }
```

Но взгляните на следующее обновленное определение записи `Car`:

```
record CarRecord(string Make, string Model, string Color);
```

В конструкторе так называемого *позиционного типа записи* определены свойства записи, а весь остальной связующий код удален. При использовании такого синтаксиса необходимо принимать во внимание три соображения. Во-первых, не разрешено применять инициализацию объектов типов записей, использующих компактный синтаксис определения, во-вторых, запись должна конструироваться со свойствами, расположенными в корректных позициях, и, в-третьих, регистр символов в свойствах конструктора точно повторяется в свойствах внутри типа записи.

Эквивалентность с типами записей

В примере класса `Car` два экземпляра `Car` создавались с одними и теми же данными. В приведенной далее проверке *может* показаться, что следующие два экземпляра класса эквивалентны:

```
Console.WriteLine($"Cars are the same? {myCar.Equals(anotherMyCar)}");
                  // Эквивалентны ли экземпляры Car?
```

Однако они не эквивалентны. Вспомните, что типы записей представляют собой специализированный вид класса, а классы являются *ссылочными типами*. Чтобы два ссылочных типа были эквивалентными, они должны указывать на тот же самый объект в памяти. В качестве дальнейшей проверки выясним, указывают ли два экземпляра `Car` на тот же самый объект:

```
Console.WriteLine($"Cars are the same reference?
{ReferenceEquals(myCar, anotherMyCar)}");
                // Указывают ли экземпляры Car на тот же самый объект?
```

Запуск программы дает приведенный ниже результат:

```
Cars are the same? False
Cars are the same? False
```

Типы записей ведут себя по-другому. Они неявно переопределяют `Equals()`, `==` и `!=`, чтобы производить результаты, как если бы экземпляры были типами значений. Взгляните на следующий код и показанные далее результаты:

```
Console.WriteLine($"CarRecords are the same?
{myCarRecord.Equals(anotherMyCarRecord)}");
          // Эквивалентны ли экземпляры CarRecord?
Console.WriteLine($"CarRecords are the same reference?
{ReferenceEquals(myCarRecord, anotherMyCarRecord)}");
          // Указывают ли экземпляры CarRecord на тот же самый объект?
Console.WriteLine($"CarRecords are the same?
  {myCarRecord == anotherMyCarRecord}");
Console.WriteLine($"CarRecords are not the same?
  {myCarRecord != anotherMyCarRecord}");
```

Вот результирующий вывод:

```
/*************** RECORDS *********************/
My car:
CarRecord { Make = Honda, Model = Pilot, Color = Blue }
Another variable for my car:
CarRecord { Make = Honda, Model = Pilot, Color = Blue }

CarRecords are the same? True
CarRecords are the same reference? False
CarRecords are the same? True
CarRecords are not the same? False
```

Обратите внимание, что записи считаются эквивалентными, невзирая на то, что переменные указывают на два разных объекта в памяти.

Копирование типов записей с использованием выражений `with`

Для типов записей присваивание экземпляра такого типа новой переменной создает указатель на ту же самую ссылку, что аналогично поведению классов. Это демонстрируется в приведенном ниже коде:

```
CarRecord carRecordCopy = anotherMyCarRecord;
Console.WriteLine("Car Record copy results");
Console.WriteLine($"CarRecords are the same?
  {carRecordCopy.Equals(anotherMyCarRecord)}");
Console.WriteLine($"CarRecords are the same? {ReferenceEquals(carRecordCopy,
  anotherMyCarRecord)}");
```

В результате запуска кода обе проверки возвращают `True`, доказывая эквивалентность по значению и по ссылке.

Для создания подлинной копии записи с модифицированным одним или большим числом свойств в версии C# 9.0 были введены *выражения* `with`. В конструкции `with` указываются любые подлежащие обновлению свойства вместе с их новыми значениями, а значения свойств, которые не были перечислены, копируются без изменений. Вот пример:

```
CarRecord ourOtherCar = myCarRecord with {Model = "Odyssey"};
Console.WriteLine("My copied car:");
Console.WriteLine(ourOtherCar.ToString());
Console.WriteLine("Car Record copy using with expression results");
                // Результаты копирования CarRecord
                // с использованием выражения with
Console.WriteLine($"CarRecords are the same?
  {ourOtherCar.Equals(myCarRecord)}");
Console.WriteLine($"CarRecords are the same?
{ReferenceEquals(ourOtherCar, myCarRecord)}");
```

В коде создается новый экземпляр типа `CarRecord` с копированием значений `Make` и `Color` экземпляра `myCarRecord` и установкой `Model` в строку `"Odyssey"`. Ниже показаны результаты выполнения кода:

```
/*************** RECORDS *********************/
My copied car:
CarRecord { Make = Honda, Model = Odyssey, Color = Blue }
Car Record copy using with expression results
CarRecords are the same? False
CarRecords are the same? False
```

С применением выражений `with` вы можете компоновать экземпляры типов записей в новые экземпляры типов записей с модифицированными значениями свойств.

На этом начальное знакомство с новыми типами записей C# 9.0 завершено. В следующей главе будут подробно исследоваться типы записей и наследование.

Резюме

Целью главы было ознакомление вас с ролью типа класса C# и нового типа записи C# 9.0. Вы видели, что классы могут иметь любое количество *конструкторов*, которые позволяют пользователю объекта устанавливать состояние объекта при его создании. В главе также было продемонстрировано несколько приемов проектирования классов (и связанных с ними ключевых слов). Ключевое слово `this` используется для получения доступа к текущему объекту. Ключевое слово `static` дает возможность определять поля и члены, привязанные к уровню класса (не объекта). Ключевое слово `const`, модификатор `readonly` и средства доступа только для инициализации позволяют определять элементы данных, которые никогда не изменяются после первоначальной установки или конструирования объекта. Типы записей являются особым видом класса, который неизменяем и при сравнении одного экземпляра типа записи с другим экземпляром того же самого типа записи ведет себя подобно типам значений.

Большая часть главы была посвящена деталям первого принципа ООП — инкапсуляции. Вы узнали о модификаторах доступа C# и роли свойств типа, о синтаксисе инициализации объектов и о частичных классах. Теперь вы готовы перейти к чтению следующей главы, в которой речь пойдет о построении семейства взаимосвязанных классов с применением наследования и полиморфизма.

ГЛАВА 6

Наследование и полиморфизм

В главе 5 рассматривался первый основной принцип объектно-ориентированного программирования (ООП) — инкапсуляция. Вы узнали, как строить отдельный четко определенный тип класса с конструкторами и разнообразными членами (полями, свойствами, методами, константами и полями только для чтения). В настоящей главе мы сосредоточим внимание на оставшихся двух принципах ООП: наследовании и полиморфизме.

Прежде всего, вы научитесь строить семейства связанных классов с применением *наследования*. Как будет показано, такая форма многократного использования кода позволяет определять в родительском классе общую функциональность, которая может быть задействована, а возможно и модифицирована в дочерних классах. В ходе изложения вы узнаете, как устанавливать *полиморфный интерфейс* в иерархиях классов, используя виртуальные и абстрактные члены, а также о роли явного приведения.

Глава завершится исследованием роли изначального родительского класса в библиотеках базовых классов .NET Core — `System.Object`.

Базовый механизм наследования

Вспомните из главы 5, что *наследование* — это аспект ООП, упрощающий повторное использование кода. Говоря более точно, встречаются две разновидности повторного использования кода: наследование (отношение "является") и модель включения/делегации (отношение "имеет"). Давайте начнем текущую главу с рассмотрения классической модели наследования, т.е. отношения "является".

Когда вы устанавливаете между классами отношение "является", то тем самым строите зависимость между двумя и более типами классов. Основная идея, лежащая в основе классического наследования, состоит в том, что новые классы могут создаваться с применением существующих классов как отправной точки. В качестве простого примера создайте новый проект консольного приложения по имени `BasicInheritance`.

Предположим, что вы спроектировали класс `Car`, который моделирует ряд базовых деталей автомобиля:

```
namespace BasicInheritance
{
  // Простой базовый класс.
  class Car
  {
    public readonly int MaxSpeed;
    private int _currSpeed;
    public Car(int max)
    {
      MaxSpeed = max;
    }
    public Car()
    {
      MaxSpeed = 55;
    }
    public int Speed
    {
      get { return _currSpeed; }
      set
      {
        _currSpeed = value;
        if (_currSpeed > MaxSpeed)
        {
          _currSpeed = MaxSpeed;
        }
      }
    }
  }
}
```

Обратите внимание, что класс `Car` использует службы инкапсуляции для управления доступом к закрытому полю `_currSpead` посредством открытого свойства по имени `Speed`. В данный момент с типом `Car` можно работать следующим образом:

```
using System;
using BasicInheritance;

Console.WriteLine("***** Basic Inheritance *****\n");
// Создать объект Car и установить максимальную и текущую скорости.
Car myCar = new Car(80) {Speed = 50};

// Вывести значение текущей скорости.
Console.WriteLine("My car is going {0} MPH", myCar.Speed);
Console.ReadLine();
```

Указание родительского класса для существующего класса

Теперь предположим, что планируется построить новый класс по имени `MiniVan`. Подобно базовому классу `Car` вы хотите определить класс `MiniVan` так, чтобы он поддерживал данные для максимальной и текущей скоростей и свойство по имени `Speed`, которое позволило бы пользователю модифицировать состояние объекта. Очевидно, что классы `Car` и `MiniVan` взаимосвязаны; фактически можно сказать, что `MiniVan` *"является"* разновидностью `Car`. Отношение "является" (формально называемое *классическим наследованием*) позволяет строить новые определения классов, которые расширяют функциональность существующих классов.

Существующий класс, который будет служить основой для нового класса, называется *базовым классом*, *суперклассом* или *родительским классом*. Роль базового класса заключается в определении всех общих данных и членов для классов, которые его расширяют. Расширяющие классы формально называются *производными* или *дочерними* классами. В языке C# для установления между классами отношения "является" применяется операция двоеточия в определении класса. Пусть вы написали новый класс `MiniVan` следующего вида:

```
namespace BasicInheritance
{
  // MiniVan "является" Car.
  sealed class MiniVan : Car
  {
  }
}
```

В текущий момент никаких членов в новом классе не определено. Так чего же мы достигли за счет наследования `MiniVan` от базового класса `Car`? Выражаясь просто, объекты `MiniVan` теперь имеют доступ ко всем открытым членам, определенным внутри базового класса.

На заметку! Несмотря на то что конструкторы обычно определяются как открытые, производный класс никогда не наследует конструкторы родительского класса. Конструкторы используются для создания только экземпляра класса, внутри которого они определены, но к ним можно обращаться в производном классе через построение цепочки вызовов конструкторов, как будет показано далее.

Учитывая отношение между этими двумя типами классов, вот как можно работать с классом `MiniVan`:

```
Console.WriteLine("***** Basic Inheritance *****\n");
...
// Создать объект MiniVan.
MiniVan myVan = new MiniVan {Speed = 10};
Console.WriteLine("My van is going {0} MPH", myVan.Speed);
Console.ReadLine();
```

Обратите внимание, что хотя в класс `MiniVan` никакие члены не добавлялись, в нем есть прямой доступ к открытому свойству `Speed` родительского класса; тем самым обеспечивается повторное использование кода. Такой подход намного лучше, чем создание класса `MiniVan`, который имеет те же самые члены, что и класс `Car`, скажем, свойство `Speed`. Дублирование кода в двух классах приводит к необходимости сопровождения двух порций кода, что определенно будет непродуктивным расходом времени.

Всегда помните о том, что наследование предохраняет инкапсуляцию, а потому следующий код вызовет ошибку на этапе компиляции, т.к. закрытые члены не могут быть доступны через объектную ссылку:

```
Console.WriteLine("***** Basic Inheritance *****\n");
...
// Создать объект MiniVan.
MiniVan myVan = new MiniVan();
myVan.Speed = 10;
```

```
Console.WriteLine("My van is going {0} MPH",
  myVan.Speed);
// Ошибка! Доступ к закрытым членам невозможен!
myVan._currSpeed = 55;
Console.ReadLine();
```

В качестве связанного примечания: даже когда класс `MiniVan` определяет собственный набор членов, он по-прежнему не будет располагать возможностью доступа к любым закрытым членам базового класса `Car`. Не забывайте, что закрытые члены доступны *только* внутри класса, в котором они определены. Например, показанный ниже метод в `MiniVan` приведет к ошибке на этапе компиляции:

```
// Класс MiniVan является производным от Car.
class MiniVan : Car
{
  public void TestMethod()
  {
    // Нормально! Доступ к открытым членам родительского
    // типа в производном типе возможен.
    Speed = 10;
    // Ошибка! Нельзя обращаться к закрытым членам
    // родительского типа из производного типа!
    _currSpeed = 10;
  }
}
```

Замечание относительно множества базовых классов

Говоря о базовых классах, важно иметь в виду, что язык C# требует, чтобы отдельно взятый класс имел в точности *один* непосредственный базовый класс. Создать тип класса, который был бы производным напрямую от двух и более базовых классов, невозможно (такой прием, поддерживаемый в неуправляемом языке C++, известен как *множественное наследование*). Попытка создать класс, для которого указаны два непосредственных родительских класса, как продемонстрировано в следующем коде, приведет к ошибке на этапе компиляции:

```
// Недопустимо! Множественное наследование
// классов в языке C# не разрешено!
class WontWork
  : BaseClassOne, BaseClassTwo
{}
```

В главе 8 вы увидите, что платформа .NET Core позволяет классу или структуре реализовывать любое количество дискретных интерфейсов. Таким способом тип C# может поддерживать несколько линий поведения, одновременно избегая сложностей, которые связаны с множественным наследованием. Применяя этот подход, можно строить развитые иерархии интерфейсов, которые моделируют сложные линии поведения (см. главу 8).

Использование ключевого слова `sealed`

Язык C# предлагает еще одно ключевое слово, `sealed`, которое предотвращает наследование. Когда класс помечен как `sealed` (запечатанный), компилятор не позволяет создавать классы, производные от него. Например, пусть вы приняли решение о том, что дальнейшее расширение класса `MiniVan` не имеет смысла:

```
// Класс Minivan не может быть расширен!
sealed class MiniVan : Car
{
}
```

Если вы или ваш коллега попытаетесь унаследовать от запечатанного класса `MiniVan`, то получите ошибку на этапе компиляции:

```
// Ошибка! Нельзя расширять класс, помеченный ключевым словом sealed!
class DeluxeMiniVan
  : MiniVan
{
}
```

Запечатывание класса чаще всего имеет наибольший смысл при проектировании обслуживающего класса. Скажем, в пространстве имен `System` определены многочисленные запечатанные классы, такие как `String`. Таким образом, как и в случае `MiniVan`, если вы попытаетесь построить новый класс, который расширял бы `System.String`, то получите ошибку на этапе компиляции:

```
// Еще одна ошибка! Нельзя расширять класс, помеченный как sealed!
class MyString
  : String
{
}
```

На заметку! В главе 4 вы узнали о том, что структуры C# всегда неявно запечатаны (см. табл. 4.3). Следовательно, создать структуру, производную от другой структуры, класс, производный от структуры, или структуру, производную от класса, невозможно. Структуры могут применяться для моделирования только отдельных, атомарных, определяемых пользователем типов. Если вы хотите задействовать отношение "является", тогда должны использовать классы.

Нетрудно догадаться, что есть многие другие детали наследования, о которых вы узнаете в оставшемся материале главы. Пока просто примите к сведению, что операция двоеточия позволяет устанавливать отношения "базовый–производный" между классами, а ключевое слово `sealed` предотвращает последующее наследование.

Еще раз о диаграммах классов Visual Studio

В главе 2 кратко упоминалось о том, что среда Visual Studio позволяет устанавливать отношения "базовый–производный" между классами визуальным образом во время проектирования. Для работы с указанным аспектом IDE-среды сначала понадобится добавить в текущий проект новый файл диаграммы классов. Выберите пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент) и щелкните на значке Class Diagram (Диаграмма классов); на рис. 6.1 видно, что файл был переименован с `ClassDiagram1.cd` на `Cars.cd`. После щелчка на кнопке Add (Добавить) отобразится пустая поверхность проектирования. Чтобы добавить типы в визуальный конструктор классов, просто перетаскивайте на эту поверхность каждый файл из окна Solution Explorer (Проводник решений). Также вспомните, что удаление элемента из визуального конструктора (путем его выбора и нажатия клавиши <Delete>) не приводит к уничтожению ассоциированного с ним исходного кода, а просто убирает элемент из поверхности конструктора. Текущая иерархия классов показана на рис. 6.2.

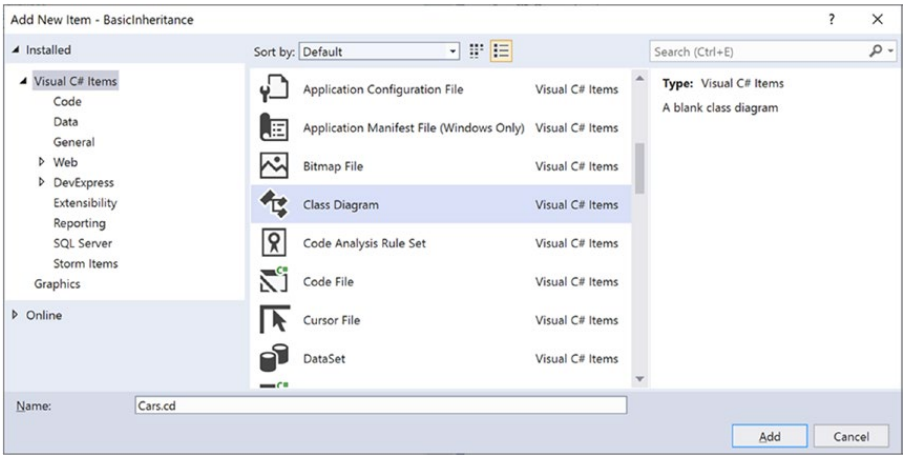

Рис. 6.1. Добавление новой диаграммы классов

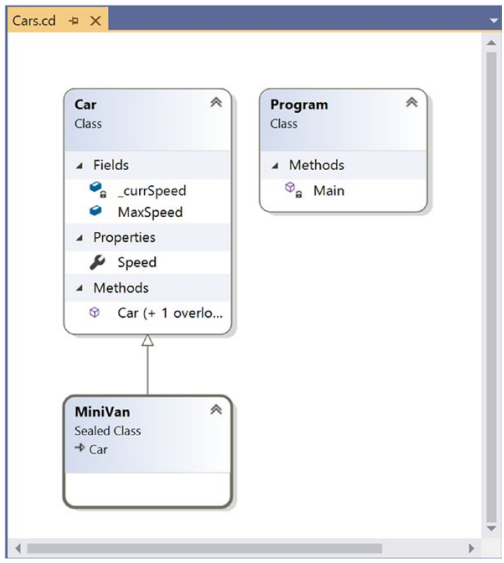

Рис. 6.2. Визуальный конструктор Visual Studio

Как говорилось в главе 2, помимо простого отображения отношений между типами внутри текущего приложения можно также создавать новые типы и наполнять их членами, применяя панель инструментов конструктора классов и окно Class Details (Детали класса).

При желании можете свободно использовать указанные визуальные инструменты во время проработки оставшихся глав книги. Однако всегда анализируйте сгенерированный код, чтобы четко понимать, что эти инструменты для вас сделали.

Второй принцип объектно-ориентированного программирования: детали наследования

Теперь, когда вы видели базовый синтаксис наследования, давайте построим более сложный пример и рассмотрим многочисленные детали построения иерархий классов. Мы снова обратимся к классу `Employee`, который был спроектирован в главе 5. Первым делом создайте новый проект консольного приложения C# по имени `Employees`.

Далее скопируйте в проект `Employees` файлы `Employee.cs`, `Employee.Core.cs` и `EmployeePayTypeEnum.cs`, созданные ранее в проекте `EmployeeApp` из главы 5.

На заметку! До выхода .NET Core, чтобы использовать файлы в проекте C#, на них необходимо было ссылаться в файле `.csproj`. В версии .NET Core все файлы из текущей структуры каталогов автоматически включаются в проект. Простого копирования нескольких файлов из другого проекта достаточно для их включения в ваш проект.

Прежде чем приступать к построению каких-то производных классов, следует уделить внимание одной детали. Поскольку первоначальный класс `Employee` был создан в проекте по имени `EmployeeApp`, он находится внутри идентично названного пространства имен .NET Core. Пространства имен подробно рассматриваются в главе 16; тем не менее, ради простоты переименуйте текущее пространство имен (в обоих файлах) на `Employees`, чтобы оно совпадало с именем нового проекта:

```
// Не забудьте изменить название пространства имен в обоих файлах C#!
namespace Employees
{
  partial class Employee
  {...}
}
```

На заметку! Если вы удалили стандартный конструктор во время внесения изменений в код класса `Employee` в главе 5, тогда снова добавьте его в класс.

Вторая деталь касается удаления любого закомментированного кода из различных итераций класса `Employee`, рассмотренных в примере главы 5.

На заметку! В качестве проверки работоспособности скомпилируйте и запустите новый проект, введя `dotnet run` в окне командной подсказки (в каталоге проекта) или нажав <Ctrl+F5> в случае использования Visual Studio. Пока что программа ничего не делает, но это позволит удостовериться в отсутствии ошибок на этапе компиляции.

Цель в том, чтобы создать семейство классов, моделирующих разнообразные типы сотрудников в компании. Предположим, что необходимо задействовать функциональность класса `Employee` при создании двух новых классов (`SalesPerson` и `Manager`). Новый класс `SalesPerson` "является" `Employee` (как и `Manager`). Вспомните, что в модели классического наследования базовые классы (вроде `Employee`) обычно применяются для определения характеристик, общих для всех наследников. Подклассы (такие как `SalesPerson` и `Manager`) расширяют общую функциональность, добавляя к ней специфическую функциональность.

В настоящем примере мы будем считать, что класс Manager расширяет Employee, сохраняя количество фондовых опционов, тогда как класс SalesPerson поддерживает хранение количества продаж. Добавьте новый файл класса (Manager.cs), в котором определен класс Manager со следующим автоматическим свойством:

```
// Менеджерам нужно знать количество их фондовых опционов.
class Manager : Employee
{
  public int StockOptions { get; set; }
}
```

Затем добавьте еще один новый файл класса (SalesPerson.cs), в котором определен класс SalesPerson с подходящим автоматическим свойством:

```
// Продавцам нужно знать количество продаж.
class SalesPerson : Employee
{
  public int SalesNumber { get; set; }
}
```

После того как отношение "является" установлено, классы SalesPerson и Manager автоматически наследуют все открытые члены базового класса Employee. В целях иллюстрации обновите операторы верхнего уровня, как показано ниже:

```
//Создание объекта подкласса и доступ к функциональности базового класса
Console.WriteLine("***** The Employee Class Hierarchy *****\n");
SalesPerson fred = new SalesPerson
{
  Age = 31, Name = "Fred", SalesNumber = 50
};
```

Вызов конструкторов базового класса с помощью ключевого слова base

В текущий момент объекты классов SalesPerson и Manager могут создаваться только с использованием "бесплатно полученного" стандартного конструктора (см. главу 5). Памятуя о данном факте, предположим, что в класс Manager добавлен новый конструктор с шестью аргументами, который вызывается следующим образом:

```
...
// Предположим, что у Manager есть конструктор с такой сигнатурой:
// (string fullName, int age, int empId,
// float currPay, string ssn, int numbOfOpts)
Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
Console.ReadLine();
```

Взглянув на список параметров, легко заметить, что большинство аргументов должно быть сохранено в переменных-членах, определенных в базовом классе Employee. Чтобы сделать это, в классе Manager можно было бы реализовать показанный ниже специальный конструктор:

```
public Manager(string fullName, int age, int empId,
               float currPay, string ssn, int numbOfOpts)
{
  // Это свойство определено в классе Manager.
  StockOptions = numbOfOpts;
```

```
  // Присвоить входные параметры, используя
  // унаследованные свойства родительского класса.
  Id = empId;
  Age = age;
  Name = fullName;
  Pay = currPay;
  PayType = EmployeePayTypeEnum.Salaried;

  // Если свойство SSN окажется доступным только для чтения,
  // тогда здесь возникнет ошибка на этапе компиляции!
  SocialSecurityNumber = ssn;
}
```

Первая проблема с таким подходом связана с тем, что если любое свойство определено как допускающее только чтение (например, свойство `SocialSecurityNumber`), то присвоить значение входного параметра `string` данному полю не удастся, как можно видеть в финальном операторе специального конструктора.

Вторая проблема заключается в том, что был косвенно создан довольно неэффективный конструктор, учитывая тот факт, что в C# стандартный конструктор базового класса вызывается автоматически перед выполнением логики конструктора производного класса, если не указано иначе. После этого момента текущая реализация имеет доступ к многочисленным открытым свойствам базового класса `Employee` для установки его состояния. Таким образом, во время создания объекта `Manager` на самом деле выполнялось восемь действий (обращения к шести унаследованным свойствам и двум конструкторам).

Для оптимизации создания объектов производного класса необходимо корректно реализовать конструкторы подкласса, чтобы они явно вызывали подходящий специальный конструктор базового класса вместо стандартного конструктора. Подобным образом можно сократить количество вызовов инициализации унаследованных членов (что уменьшит время обработки). Первым делом обеспечьте наличие в родительском классе `Employee` следующего конструктора с шестью аргументами:

```
// Добавление в базовый класс Employee.
public Employee(string name, int age, int id, float pay, string empSsn,
  EmployeePayTypeEnum payType)
{
  Name = name;
  Id = id;
  Age = age;
  Pay = pay;
  SocialSecurityNumber = empSsn;
  PayType = payType;
}
```

Модифицируйте специальный конструктор в классе `Manager`, чтобы вызвать конструктор `Employee` с применением ключевого слова `base`:

```
public Manager(string fullName, int age, int empId,
  float currPay, string ssn, int numbOfOpts)
  : base(fullName, age, empId, currPay, ssn,
       EmployeePayTypeEnum.Salaried)
{
  // Это свойство определено в классе Manager.
  StockOptions = numbOfOpts;
}
```

Здесь ключевое слово base ссылается на сигнатуру конструктора (подобно синтаксису, используемому для объединения конструкторов одиночного класса в цепочку через ключевое слово this, как обсуждалось в главе 5), что всегда указывает производному конструктору на необходимость передачи данных конструктору непосредственного родительского класса. В рассматриваемой ситуации явно вызывается конструктор с шестью параметрами, определенный в Employee, что избавляет от излишних обращений во время создания объекта дочернего класса. Кроме того, в класс Manager добавлена особая линия поведения, которая заключается в том, что тип оплаты всегда устанавливается в Salaried. Специальный конструктор класса SalesPerson выглядит почти идентично, но только тип оплаты устанавливается в Commission:

```
// В качестве общего правила запомните, что все подклассы должны
// явно вызывать подходящий конструктор базового класса.
public SalesPerson(string fullName, int age, int empId,
  float currPay, string ssn, int numbOfSales)
  : base(fullName, age, empId, currPay, ssn,
       EmployeePayTypeEnum.Commission)
{
  // Это принадлежит нам!
  SalesNumber = numbOfSales;
}
```

На заметку! Ключевое слово base можно применять всякий раз, когда подкласс желает обратиться к открытому или защищенному члену, определенному в родительском классе. Использование этого ключевого слова не ограничивается логикой конструктора. Вы увидите примеры применения ключевого слова base в подобной манере позже в главе при рассмотрении полиморфизма.

Наконец, вспомните, что после добавления к определению класса специального конструктора стандартный конструктор молча удаляется. Следовательно, не забудьте переопределить стандартный конструктор для классов SalesPerson и Manager. Вот пример:

```
// Аналогичным образом переопределите стандартный
// конструктор также и в классе Manager.
public SalesPerson() {}
```

Хранение секретов семейства: ключевое слово protected

Как вы уже знаете, открытые элементы напрямую доступны отовсюду, в то время как закрытые элементы могут быть доступны только в классе, где они определены. Вспомните из главы 5, что C# опережает многие другие современные объектные языки и предоставляет дополнительное ключевое слово для определения доступности членов — protected (защищенный).

Когда базовый класс определяет защищенные данные или защищенные члены, он устанавливает набор элементов, которые могут быть непосредственно доступны любому наследнику. Если вы хотите разрешить дочерним классам SalesPerson и Manager напрямую обращаться к разделу данных, который определен в Employee, то модифицируйте исходный класс Employee (в файле EmployeeCore.cs), как показано ниже:

```
// Защищенные данные состояния.
partial class Employee
{
  //Производные классы теперь могут иметь прямой доступ к этой информации
  protected string EmpName;
  protected int EmpId;
  protected float CurrPay;
  protected int EmpAge;
  protected string EmpSsn;
  protected EmployeePayTypeEnum EmpPayType;
  ...
}
```

На заметку! По соглашению защищенные члены именуются в стиле Pascal (`EmpName`), а не в "верблюжьем" стиле с подчеркиванием (`_empName`). Это не является требованием языка, но представляет собой распространенный стиль написания кода. Если вы решите обновить имена, как было сделано здесь, тогда не забудьте переименовать все поддерживающие методы в свойствах, чтобы они соответствовали защищенным свойствам с именами в стиле Pascal.

Преимущество определения защищенных членов в базовом классе заключается в том, что производным классам больше не придется обращаться к данным косвенно, используя открытые методы и свойства. Разумеется, подходу присущ и недостаток: когда производный класс имеет прямой доступ к внутренним данным своего родителя, то есть вероятность непредумышленного обхода существующих бизнес-правил, которые реализованы внутри открытых свойств. Определяя защищенные члены, вы создаете уровень доверия между родительским классом и дочерним классом, т.к. компилятор не будет перехватывать какие-либо нарушения бизнес-правил, предусмотренных для типа.

Наконец, имейте в виду, что с точки зрения пользователя объекта защищенные данные расцениваются как закрытые (поскольку пользователь находится "снаружи" семейства). По указанной причине следующий код недопустим:

```
// Ошибка! Доступ к защищенным данным из клиентского кода невозможен!
Employee emp = new Employee();
emp.empName = "Fred";
```

На заметку! Несмотря на то что защищенные поля данных могут нарушить инкапсуляцию, определять защищенные методы вполне безопасно (и полезно). При построении иерархий классов обычно приходится определять набор методов, которые предназначены для применения только производными типами, но не внешним миром.

Добавление запечатанного класса

Вспомните, что *запечатанный* класс не может быть расширен другими классами. Как уже упоминалось, такой прием чаще всего используется при проектировании обслуживающих классов. Тем не менее, при построении иерархий классов вы можете обнаружить, что определенная ветвь в цепочке наследования нуждается в "отсечении", т.к. дальнейшее ее расширение не имеет смысла. В качестве примера предположим, что вы добавили в приложение еще один класс (`PtSalesPerson`), который расширяет существующий тип `SalesPerson`. Текущее обновление показано на рис. 6.3.

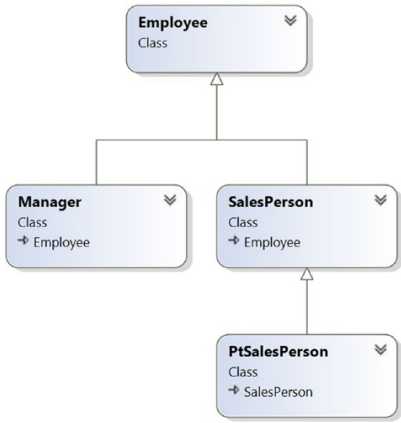

Рис. 6.3. Класс `PtSalesPerson`

Класс `PtSalesPerson` представляет продавца, который работает на условиях частичной занятости. В качестве варианта скажем, что нужно гарантировать отсутствие возможности создания подкласса `PtSalesPerson`. Чтобы предотвратить наследование от класса, необходимо применить ключевое слово `sealed`:

```
sealed class PtSalesPerson : SalesPerson
{
  public PtSalesPerson(string fullName, int age, int empId,
    float currPay, string ssn, int numbOfSales)
    : base(fullName, age, empId, currPay, ssn, numbOfSales)
  {
  }
  // Остальные члены класса...
}
```

Наследование с типами записей (нововведение в версии 9.0)

Появившиеся в версии C# 9.0 типы записей также поддерживают наследование. Чтобы выяснить как, отложите пока свою работу над проектом `Employees` и создайте новый проект консольного приложения по имени `RecordInheritance`. Добавьте в него два файла с именами `Car.cs` и `MiniVan.cs`, содержащими следующие определения записей:

```
// Car.cs
namespace RecordInheritance
{
  // Тип записи Car
  public record Car
  {
    public string Make { get; init; }
    public string Model { get; init; }
    public string Color { get; init; }

    public Car(string make, string model, string color)
    {
```

```
      Make = make;
      Model = model;
      Color = color;
    }
  }
}
// MiniVan.cs
namespace RecordInheritance
{
  // Тип записи MiniVan
  public sealed record MiniVan : Car
  {
    public int Seating { get; init; }
    public MiniVan(string make, string model, string color, int seating)
      : base(make, model, color)
    {
      Seating = seating;
    }
  }
}
```

Обратите внимание, что между примерами использования типов записей и предшествующими примерами применения классов нет большой разницы. Модификатор доступа `protected` для свойств и методов ведет себя аналогично, а модификатор доступа `sealed` для типа записи запрещает другим типам записей быть производными от запечатанных типов записей. Вы также обнаружите работу с унаследованными типами записей в оставшихся разделах главы. Причина в том, что типы записей — это всего лишь особый вид неизменяемого класса (как объяснялось в главе 5).

Вдобавок типы записей включают неявные приведения к своим базовым классам, что демонстрируется в коде ниже:

```
using System;
using RecordInheritance;

Console.WriteLine("Record type inheritance!");

Car c = new Car("Honda","Pilot","Blue");
MiniVan m = new MiniVan("Honda", "Pilot", "Blue",10);
Console.WriteLine($"Checking MiniVan is-a Car:{m is Car}");
                  // Проверка, является ли MiniVan типом Car
```

Как и можно было ожидать, проверка того, что `m` является `Car`, возвращает `true`, как видно в следующем выводе:

```
Record type inheritance!
Checking minvan is-a car:True
```

Важно отметить, что хотя типы записей представляют собой специализированные классы, вы не можете организовывать перекрестное наследование между классами и записями. Другими словами, классы нельзя наследовать от типов записей, а типы записей не допускается наследовать от классов. Взгляните на приведенный далее код; последние два примера не скомпилируются:

```
namespace RecordInheritance
{
  public class TestClass { }
  public record TestRecord { }
```

```
  // Классы не могут быть унаследованы от записей.
  // public class Test2 : TestRecord { }

  // Записи не могут быть унаследованы от классов.
  // public record Test2 : TestClass { }
}
```

Наследование также работает с позиционными типами записей. Создайте в своем проекте новый файл по имени `PositionalRecordTypes.cs` и поместите в него следующий код:

```
namespace RecordInheritance
{
  public record PositionalCar (string Make, string Model, string Color);
  public record PositionalMiniVan (string Make, string Model, string Color)
    : PositionalCar(Make, Model, Color);
}
```

Добавьте к операторам верхнего уровня показанный ниже код, с помощью которого можно подтвердить то, что вам уже известно: позиционные типы записей работают точно так же, как типы записей.

```
PositionalCar pc = new PositionalCar("Honda", "Pilot", "Blue");
PositionalMiniVan pm = new PositionalMiniVan("Honda", "Pilot", "Blue", 10);
Console.WriteLine($"Checking PositionalMiniVan is-a PositionalCar:
 {pm is PositionalCar}");
```

Эквивалентность с унаследованными типами записей

Вспомните из главы 5, что для определения эквивалентности типы записей используют семантику значений. Еще одна деталь относительно типов записей связана с тем, что *тип* записи является частью соображения, касающегося эквивалентности. Скажем, взгляните на следующие тривиальные примеры:

```
public record MotorCycle(string Make, string Model);
public record Scooter(string Make, string Model) :
MotorCycle(Make,Model);
```

Игнорируя тот факт, что унаследованные классы обычно расширяют базовые классы, в приведенных простых примерах определяются два разных типа записей, которые имеют те же самые свойства. В случае создания экземпляров с одинаковыми значениями для свойств они не пройдут проверку на предмет эквивалентности из-за того, что принадлежат разным типам. В качестве примера рассмотрим показанный далее код и результаты его выполнения:

```
MotorCycle mc = new MotorCycle("Harley","Lowrider");
Scooter sc = new Scooter("Harley", "Lowrider");
Console.WriteLine($"MotorCycle and Scooter are equal:
{Equals(mc,sc)}");
```

Вот вывод:

```
Record type inheritance!
MotorCycle and Scooter are equal: False
```

Реализация модели включения/делегации

Вам уже известно, что повторное использование кода встречается в двух видах. Только что было продемонстрировано классическое отношение "является". Перед тем, как мы начнем исследование третьего принципа ООП (полиморфизма), давайте взглянем на отношение "имеет" (также известное как *модель включения/делегации* или *агрегация*). Возвратитесь к проекту `Employees` и создайте новый файл по имени `BenefitPackage.cs`. Поместите в него следующий код, моделирующий пакет льгот для сотрудников:

```
namespace Employees
{
  // Этот новый тип будет функционировать как включаемый класс.
  class BenefitPackage
  {
    // Предположим, что есть другие члены, представляющие
    // медицинские/стоматологические программы и т.п.
    public double ComputePayDeduction()
    {
      return 125.0;
    }
  }
}
```

Очевидно, что было бы довольно странно устанавливать отношение "является" между классом `BenefitPackage` и типами сотрудников. (Разве сотрудник "является" пакетом льгот? Вряд ли.) Однако должно быть ясно, что какое-то отношение между ними должно быть установлено. Короче говоря, нужно выразить идею о том, что каждый сотрудник "имеет" пакет льгот. Для этого можно модифицировать определение класса `Employee` следующим образом:

```
// Теперь сотрудники имеют льготы.
partial class Employee
{
  // Содержит объект BenefitPackage.
  protected BenefitPackage EmpBenefits = new BenefitPackage();
  ...
}
```

На данной стадии вы имеете объект, который благополучно содержит в себе другой объект. Тем не менее, открытие доступа к функциональности содержащегося объекта внешнему миру требует делегации. *Делегация* — просто действие по добавлению во включающий класс открытых членов, которые работают с функциональностью содержащегося внутри объекта.

Например, вы могли бы изменить класс `Employee` так, чтобы он открывал доступ к включенному объекту `EmpBenefits` с применением специального свойства, а также использовать его функциональность внутренне посредством нового метода по имени `GetBenefitCost()`:

```
partial class Employee
{
  // Содержит объект BenefitPackage.
  protected BenefitPackage EmpBenefits = new BenefitPackage();
```

```
  // Открывает доступ к некоторому поведению, связанному со льготами.
  public double GetBenefitCost()
    => EmpBenefits.ComputePayDeduction();

  // Открывает доступ к объекту через специальное свойство.
  public BenefitPackage Benefits
  {
    get { return EmpBenefits; }
    set { EmpBenefits = value; }
  }
}
```

В показанном ниже обновленном коде верхнего уровня обратите внимание на взаимодействие с внутренним типом `BenefitsPackage`, который определен в типе `Employee`:

```
Console.WriteLine("***** The Employee Class Hierarchy *****\n");
...
Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
double cost = chucky.GetBenefitCost();
Console.WriteLine($"Benefit Cost: {cost}");
Console.ReadLine();
```

Определения вложенных типов

В главе 5 кратко упоминалась концепция вложенных типов, которая является развитием рассмотренного выше отношения "имеет". В C# (а также в других языках .NET) допускается определять тип (перечисление, класс, интерфейс, структуру или делегат) прямо внутри области действия класса либо структуры. В таком случае вложенный (или "внутренний") тип считается членом охватывающего (или "внешнего") типа, и в глазах исполняющей системы им можно манипулировать как любым другим членом (полем, свойством, методом и событием). Синтаксис, применяемый для вложения типа, достаточно прост:

```
public class OuterClass
{
  // Открытый вложенный тип может использоваться кем угодно.
  public class PublicInnerClass {}

  // Закрытый вложенный тип может использоваться
  // только членами включающего класса.
  private class PrivateInnerClass {}
}
```

Хотя синтаксис довольно ясен, ситуации, в которых это может понадобиться, не настолько очевидны. Для того чтобы понять данный прием, рассмотрим характерные черты вложенных типов.

- Вложенные типы позволяют получить полный контроль над уровнем доступа внутреннего типа, потому что они могут быть объявлены как закрытые (вспомните, что невложенные классы нельзя объявлять с ключевым словом `private`).
- Поскольку вложенный тип является членом включающего класса, он может иметь доступ к закрытым членам этого включающего класса.
- Часто вложенный тип полезен только как вспомогательный для внешнего класса и не предназначен для использования во внешнем мире.

Когда тип включает в себя другой тип класса, он может создавать переменные-члены этого типа, как в случае любого другого элемента данных. Однако если с вложенным типом нужно работать за пределами включающего типа, тогда его придется уточнять именем включающего типа. Взгляните на приведенный ниже код:

```
// Создать и использовать объект открытого вложенного класса. Нормально!
OuterClass.PublicInnerClass inner;
inner = new OuterClass.PublicInnerClass();

// Ошибка на этапе компиляции! Доступ к закрытому вложенному
// классу невозможен!
OuterClass.PrivateInnerClass inner2;
inner2 = new OuterClass.PrivateInnerClass();
```

Для применения такой концепции в примере с сотрудниками предположим, что определение `BenefitPackage` теперь вложено непосредственно в класс `Employee`:

```
partial class Employee
{
  public class BenefitPackage
  {
    // Предположим, что есть другие члены, представляющие
    // медицинские/стоматологические программы и т.д.
    public double ComputePayDeduction()
    {
      return 125.0;
    }
  }
  ...
}
```

Процесс вложения может распространяться настолько "глубоко", насколько требуется. Например, пусть необходимо создать перечисление по имени `BenefitPackageLevel`, документирующее разнообразные уровни льгот, которые может выбирать сотрудник. Чтобы программно обеспечить тесную связь между типами `Employee`, `BenefitPackage` и `BenefitPackageLevel`, перечисление можно вложить следующим образом:

```
// В класс Employee вложен класс BenefitPackage.
public partial class Employee
{
  // В класс BenefitPackage вложено перечисление BenefitPackageLevel.
  public class BenefitPackage
  {
    public enum BenefitPackageLevel
    {
      Standard, Gold, Platinum
    }

    public double ComputePayDeduction()
    {
      return 125.0;
    }
  }
  ...
}
```

Вот как приходится использовать перечисление `BenefitPackageLevel` из-за отношений вложения:

```
static void Main(string[] args)
{
  ...
  // Определить уровень льгот.
  Employee.BenefitPackage.BenefitPackageLevel myBenefitLevel =
    Employee.BenefitPackage.BenefitPackageLevel.Platinum;
  Console.ReadLine();
}
```

Итак, к настоящему моменту вы ознакомились с несколькими ключевыми словами (и концепциями), которые позволяют строить иерархии типов, связанных посредством классического наследования, включения и вложения. Не беспокойтесь, если пока еще не все детали ясны. На протяжении оставшихся глав книги будет построено немало иерархий. А теперь давайте перейдем к исследованию последнего принципа ООП — полиморфизма.

Третий принцип объектно-ориентированного программирования: поддержка полиморфизма в C#

Вспомните, что в базовом классе `Employee` определен метод по имени `GiveBonus()`, который первоначально был реализован так (до его обновления с целью использования шаблона свойств):

```
public partial class Employee
{
  public void GiveBonus(float amount) => _currPay += amount;
  ...
}
```

Поскольку метод `GiveBonus()` был определен с ключевым словом `public`, бонусы можно раздавать продавцам и менеджерам (а также продавцам с частичной занятостью):

```
Console.WriteLine("***** The Employee Class Hierarchy *****\n");

// Выдать каждому сотруднику бонус?
Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
chucky.GiveBonus(300);
chucky.DisplayStats();
Console.WriteLine();

SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
fran.GiveBonus(200);
fran.DisplayStats();
Console.ReadLine();
```

Проблема с текущим проектным решением заключается в том, что открыто унаследованный метод `GiveBonus()` функционирует идентично для всех подклассов. В идеале при подсчете бонуса для штатного продавца и частично занятого продавца должно приниматься во внимание количество продаж. Возможно, менеджеры вместе

с денежным вознаграждением должны получать дополнительные фондовые опционы. Учитывая это, вы однажды столкнетесь с интересным вопросом: "Как сделать так, чтобы связанные типы реагировали по-разному на один и тот же запрос?". Попробуем найти на него ответ.

Использование ключевых слов `virtual` и `override`

Полиморфизм предоставляет подклассу способ определения собственной версии метода, определенного в его базовом классе, с применением процесса, который называется *переопределением метода*. Чтобы модернизировать текущее проектное решение, необходимо понимать смысл ключевых слов `virtual` и `override`. Если базовый класс желает определить метод, который *может быть* (но не обязательно) переопределен в подклассе, то он должен пометить его ключевым словом `virtual`:

```
partial class Employee
{
  // Теперь этот метод может быть переопределен в производном классе.
  public virtual void GiveBonus(float amount)
  {
    Pay += amount;
  }
  ...
}
```

На заметку! Методы, помеченные ключевым словом `virtual`, называются виртуальными методами.

Когда подкласс желает изменить реализацию деталей виртуального метода, он прибегает к помощи ключевого слова `override`. Например, классы `SalesPerson` и `Manager` могли бы переопределять метод `GiveBonus()`, как показано ниже (предположим, что класс `PtSalesPerson` не будет переопределять `GiveBonus()`, а потому просто наследует его версию из `SalesPerson`):

```
using System;
class SalesPerson : Employee
{
  ...
  // Бонус продавца зависит от количества продаж.
  public override void GiveBonus(float amount)
  {
    int salesBonus = 0;
    if (SalesNumber >= 0 && SalesNumber <= 100)
      salesBonus = 10;
    else
    {
      if (SalesNumber >= 101 && SalesNumber <= 200)
        salesBonus = 15;
      else
        salesBonus = 20;
    }
    base.GiveBonus(amount * salesBonus);
  }
}
```

```
class Manager : Employee
{
  ...
  public override void GiveBonus(float amount)
  {
    base.GiveBonus(amount);
    Random r = new Random();
    StockOptions += r.Next(500);
  }
}
```

Обратите внимание, что каждый переопределенный метод может задействовать стандартное поведение посредством ключевого слова `base`.

Таким образом, полностью повторять реализацию логики метода `GiveBonus()` вовсе не обязательно, а взамен можно повторно использовать (и расширять) стандартное поведение родительского класса.

Также предположим, что текущий метод `DisplayStats()` класса `Employee` объявлен виртуальным:

```
public virtual void DisplayStats()
{
  Console.WriteLine("Name: {0}", Name);
  Console.WriteLine("Id: {0}", Id);
  Console.WriteLine("Age: {0}", Age);
  Console.WriteLine("Pay: {0}", Pay);
  Console.WriteLine("SSN: {0}", SocialSecurityNumber);
}
```

Тогда каждый подкласс может переопределять метод `DisplayStats()` с целью отображения количества продаж (для продавцов) и текущих фондовых опционов (для менеджеров). Например, рассмотрим версию метода `DisplayStats()` из класса `Manager` (класс `SalesPerson` реализовывал бы метод `DisplayStats()` в похожей манере, выводя на консоль количество продаж):

```
// Manager.cs
public override void DisplayStats()
{
  base.DisplayStats();
  // Вывод количества фондовых опционов
  Console.WriteLine("Number of Stock Options: {0}", StockOptions);
}

// SalesPerson.cs
public override void DisplayStats()
{
  base.DisplayStats();
  // Вывод количества продаж
  Console.WriteLine("Number of Sales: {0}", SalesNumber);
}
```

Теперь, когда каждый подкласс может истолковывать эти виртуальные методы значащим для него образом, их экземпляры ведут себя как более независимые сущности:

```
Console.WriteLine("***** The Employee Class Hierarchy *****\n");

// Лучшая система бонусов!
Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
chucky.GiveBonus(300);
chucky.DisplayStats();
Console.WriteLine();

SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
fran.GiveBonus(200);
fran.DisplayStats();
Console.ReadLine();
```

Вот результат тестового запуска приложения в его текущем виде:

```
***** The Employee Class Hierarchy *****

Name: Chucky
ID: 92
Age: 50
Pay: 100300
SSN: 333-23-2322
Number of Stock Options: 9337

Name: Fran
ID: 93
Age: 43
Pay: 5000
SSN: 932-32-3232
Number of Sales: 31
```

Переопределение виртуальных членов с помощью Visual Studio/Visual Studio Code

Вы наверняка заметили, что при переопределении члена класса приходится вспоминать тип каждого параметра, не говоря уже об имени метода и соглашениях по передаче параметров (`ref`, `out` и `params`). В Visual Studio и Visual Studio Code доступно полезное средство IntelliSense, к которому можно обращаться при переопределении виртуального члена. Если вы наберете слово `override` внутри области действия типа класса (и затем нажмете клавишу пробела), то IntelliSense автоматически отобразит список всех допускающих переопределение членов родительского класса, исключая уже переопределенные методы.

Если вы выберете член и нажмете клавишу <Enter>, то IDE-среда отреагирует автоматическим заполнением заглушки метода. Обратите внимание, что вы также получаете оператор кода, который вызывает родительскую версию виртуального члена (можете удалить эту строку, если она не нужна). Например, при использовании описанного приема для переопределения метода `DisplayStats()` вы обнаружите следующий автоматически сгенерированный код:

```
public override void DisplayStats()
{
  base.DisplayStats();
}
```

Запечатывание виртуальных членов

Вспомните, что к типу класса можно применить ключевое слово `sealed`, чтобы предотвратить расширение его поведения другими типами через наследование. Ранее класс `PtSalesPerson` был запечатан на основе предположения о том, что разработчикам не имеет смысла дальше расширять эту линию наследования.

Следует отметить, что временами желательно не запечатывать класс целиком, а просто предотвратить переопределение некоторых виртуальных методов в производных типах. В качестве примера предположим, что вы не хотите, чтобы продавцы с частичной занятостью получали специальные бонусы. Предотвратить переопределение виртуального метода `GiveBonus()` в классе `PtSalesPerson` можно, запечатав данный метод в классе `SalesPerson`:

```
// Класс SalesPerson запечатал метод GiveBonus()!
class SalesPerson : Employee
{
  ...
  public override sealed void GiveBonus(float amount)
  {
    ...
  }
}
```

Здесь класс `SalesPerson` на самом деле переопределяет виртуальный метод `GiveBonus()`, определенный в `Employee`, но явно помечает его как `sealed`. Таким образом, попытка переопределения метода `GiveBonus()` в классе `PtSalesPerson` приведет к ошибке на этапе компиляции:

```
sealed class PtSalesPerson : SalesPerson
{
  ...
  // Ошибка на этапе компиляции! Переопределять этот метод
  // в классе PtSalesPerson нельзя, т.к. он был запечатан.
  public override void GiveBonus(float amount)
  {
  }
}
```

Абстрактные классы

В настоящий момент базовый класс `Employee` спроектирован так, что поставляет различные данные-члены своим наследникам, а также предлагает два виртуальных метода (`GiveBonus()` и `DisplayStats()`), которые могут быть переопределены в наследниках. Хотя все это замечательно, у такого проектного решения имеется один весьма странный побочный эффект: создавать экземпляры базового класса `Employee` можно напрямую:

```
// Что это будет означать?
Employee X = new Employee();
```

В нашем примере базовый класс `Employee` служит единственной цели — определять общие члены для всех подклассов. По всем признакам мы не намерены позволять кому-либо создавать непосредственные экземпляры типа `Employee`, т.к. он концептуально чересчур общий. Например, если кто-то заявит, что он сотрудник, то

тут же возникнет вопрос: сотрудник какого *рода* (консультант, инструктор, административный работник, литературный редактор, советник в правительстве)?

Учитывая, что многие базовые классы имеют тенденцию быть довольно расплывчатыми сущностями, намного более эффективным проектным решением для данного примера будет предотвращение возможности непосредственного создания в коде нового объекта `Employee`. В C# цели можно добиться за счет использования ключевого слова `abstract` в определении класса, создавая в итоге *абстрактный базовый класс*:

```
// Превращение класса Employee в абстрактный для
// предотвращения прямого создания его экземпляров.
abstract partial class Employee
{
  ...
}
```

Теперь попытка создания экземпляра класса `Employee` приводит к ошибке на этапе компиляции:

```
// Ошибка! Нельзя создавать экземпляр абстрактного класса!
Employee X = new Employee();
```

Рис. 6.4. Иерархия классов, представляющих сотрудников

Определение класса, экземпляры которого нельзя создавать напрямую, на первый взгляд может показаться странным. Однако вспомните, что базовые классы (абстрактные или нет) полезны тем, что содержат все общие данные и функциональность для производных типов. Такая форма абстракции дает возможность считать, что "идея" сотрудника является полностью допустимой, просто это не конкретная сущность. Кроме того, необходимо понимать, что хотя *непосредственно* создавать экземпляры абстрактного класса невозможно, они все равно появляются в памяти при создании экземпляров производных классов. Таким образом, для абстрактных классов вполне нормально (и общепринято) определять любое количество конструкторов, которые вызываются *косвенно*, когда выделяется память под экземпляры производных классов.

На данной стадии у нас есть довольно интересная иерархия сотрудников. Мы добавим чуть больше функциональности к приложению позже, при рассмотрении правил приведения типов C#. А пока на рис. 6.4 представлено текущее проектное решение.

Полиморфные интерфейсы

Когда класс определен как абстрактный базовый (посредством ключевого слова `abstract`), в нем может определяться любое число *абстрактных членов*. Абстрактные члены могут применяться везде, где требуется определить член, который *не* предоставляет стандартной реализации, но *должен* приниматься во внимание каждым производным классом. Тем самым вы навязываете *полиморфный интерфейс* каждому наследнику, оставляя им задачу реализации конкретных деталей абстрактных методов.

Выражаясь упрощенно, полиморфный интерфейс абстрактного базового класса просто ссылается на его набор виртуальных и абстрактных методов. На самом деле это намного интереснее, чем может показаться на первый взгляд, поскольку данная характерная черта ООП позволяет строить легко расширяемые и гибкие приложения. В целях иллюстрации мы реализуем (и слегка модифицируем) иерархию фигур, кратко описанную в главе 5 во время обзора основных принципов ООП. Для начала создадим новый проект консольного приложения C# по имени `Shapes`.

На рис. 6.5 обратите внимание на то, что типы `Hexagon` и `Circle` расширяют базовый класс `Shape`. Как и любой базовый класс, `Shape` определяет набор членов (в данном случае свойство `PetName` и метод `Draw()`), общих для всех наследников.

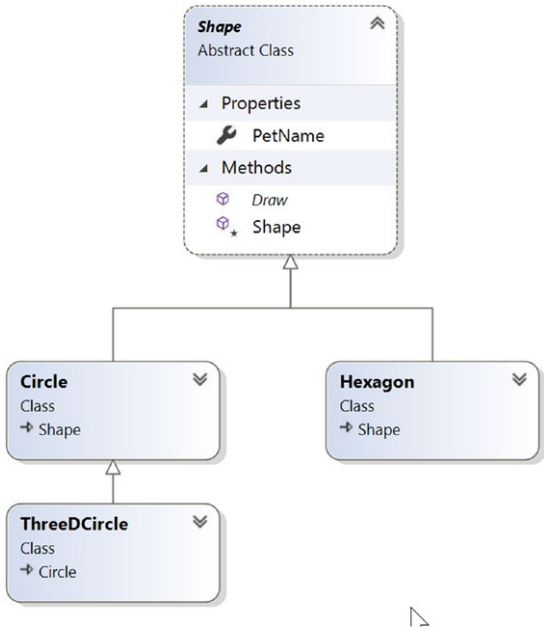

Рис. 6.5. Иерархия классов, представляющих фигуры

Во многом подобно иерархии классов для сотрудников вы должны иметь возможность запретить создание экземпляров класса `Shape` напрямую, потому что он представляет слишком абстрактную концепцию. Чтобы предотвратить непосредственное создание экземпляров класса `Shape`, его можно определить как абстрактный класс.

К тому же с учетом того, что производные типы должны уникальным образом реагировать на вызов метода Draw(), пометьте его как virtual и определите стандартную реализацию. Важно отметить, что конструктор помечен как protected, поэтому его можно вызывать только в производных классах.

```
// Абстрактный базовый класс иерархии.
abstract class Shape
{
  protected Shape(string name = "NoName")
  { PetName = name; }
  public string PetName { get; set; }
  // Единственный виртуальный метод.
  public virtual void Draw()
  {
    Console.WriteLine("Inside Shape.Draw()");
  }
}
```

Обратите внимание, что виртуальный метод Draw() предоставляет стандартную реализацию, которая просто выводит на консоль сообщение, информирующее о факте вызова метода Draw() из базового класса Shape. Теперь вспомните, что когда метод помечен ключевым словом virtual, он поддерживает стандартную реализацию, которую автоматически наследуют все производные типы. Если дочерний класс так решит, то он *может* переопределить такой метод, но он не *обязан* это делать. Рассмотрим показанную ниже реализацию типов Circle и Hexagon:

```
// В классе Circle метод Draw() НЕ переопределяется.
class Circle : Shape
{
  public Circle() {}
  public Circle(string name) : base(name){}
}
// В классе Hexagon метод Draw() переопределяется.
class Hexagon : Shape
{
  public Hexagon() {}
  public Hexagon(string name) : base(name){}
  public override void Draw()
  {
    Console.WriteLine("Drawing {0} the Hexagon", PetName);
  }
}
```

Полезность абстрактных методов становится совершенно ясной, как только вы снова вспомните, что подклассы *никогда не обязаны* переопределять виртуальные методы (как в случае Circle). Следовательно, если создать экземпляры типов Hexagon и Circle, то обнаружится, что Hexagon знает, как правильно "рисовать" себя (или, по крайней мере, выводить на консоль подходящее сообщение). Тем не менее, реакция Circle порядком сбивает с толку.

```
Console.WriteLine("***** Fun with Polymorphism *****\n");

Hexagon hex = new Hexagon("Beth");
hex.Draw();
Circle cir = new Circle("Cindy");
```

```
// Вызывает реализацию базового класса!
cir.Draw();
Console.ReadLine();
```

Взгляните на вывод предыдущего кода:

```
***** Fun with Polymorphism *****

Drawing Beth the Hexagon
Inside Shape.Draw()
```

Очевидно, что это не самое разумное проектное решение для текущей иерархии. Чтобы вынудить каждый дочерний класс переопределять метод `Draw()`, его можно определить как абстрактный метод класса `Shape`, т.е. какая-либо стандартная реализация вообще не предлагается. Для пометки метода как абстрактного в C# используется ключевое слово `abstract`. Обратите внимание, что абстрактные методы не предоставляют никакой реализации:

```
abstract class Shape
{
  // Вынудить все дочерние классы определять способ своей визуализации.
  public abstract void Draw();
  ...
}
```

На заметку! Абстрактные методы могут быть определены только в абстрактных классах, иначе возникнет ошибка на этапе компиляции.

Методы, помеченные как `abstract`, являются чистым протоколом. Они просто определяют имя, возвращаемый тип (если есть) и набор параметров (при необходимости). Здесь абстрактный класс `Shape` информирует производные типы о том, что у него есть метод по имени `Draw()`, который не принимает аргументов и ничего не возвращает. О необходимых деталях должен позаботиться производный класс.

С учетом сказанного метод `Draw()` в классе `Circle` теперь должен быть обязательно переопределен. В противном случае `Circle` также должен быть абстрактным классом и декорироваться ключевым словом `abstract` (что очевидно не подходит в настоящем примере). Вот изменения в коде:

```
// Если не реализовать здесь абстрактный метод Draw(), то Circle
// также должен считаться абстрактным и быть помечен как abstract!
class Circle : Shape
{
  public Circle() {}
  public Circle(string name) : base(name) {}
  public override void Draw()
  {
    Console.WriteLine("Drawing {0} the Circle", PetName);
  }
}
```

Итак, теперь можно предполагать, что любой класс, производный от `Shape`, действительно имеет уникальную версию метода `Draw()`. Для демонстрации полной картины полиморфизма рассмотрим следующий код:

```
Console.WriteLine("***** Fun with Polymorphism *****\n");

// Создать массив совместимых с Shape объектов.
Shape[] myShapes = {new Hexagon(), new Circle(), new Hexagon("Mick"),
  new Circle("Beth"), new Hexagon("Linda")};

// Пройти в цикле по всем элементам и взаимодействовать
// с полиморфным интерфейсом.
foreach (Shape s in myShapes)
{
  s.Draw();
}
Console.ReadLine();
```

Ниже показан вывод, выдаваемый этим кодом:

```
***** Fun with Polymorphism *****

Drawing NoName the Hexagon
Drawing NoName the Circle
Drawing Mick the Hexagon
Drawing Beth the Circle
Drawing Linda the Hexagon
```

Данный код иллюстрирует полиморфизм в чистом виде. Хотя *напрямую* создавать экземпляры абстрактного базового класса (`Shape`) невозможно, с помощью абстрактной базовой переменной допускается хранить ссылки на объекты любого подкласса. Таким образом, созданный массив объектов `Shape` способен хранить объекты классов, производных от базового класса `Shape` (попытка добавления в массив объектов, несовместимых с `Shape`, приведет к ошибке на этапе компиляции).

С учетом того, что все элементы в массиве `myShapes` на самом деле являются производными от `Shape`, вы знаете, что все они поддерживают один и тот же "полиморфный интерфейс" (или, говоря проще, все они имеют метод `Draw()`). Во время итерации по массиву ссылок `Shape` исполняющая система самостоятельно определяет лежащий в основе тип элемента. В этот момент и вызывается корректная версия метода `Draw()`.

Такой прием также делает простым безопасное расширение текущей иерархии. Например, пусть вы унаследовали от абстрактного базового класса `Shape` дополнительные классы (`Triangle`, `Square` и т.д.). Благодаря полиморфному интерфейсу код внутри цикла `foreach` не потребует никаких изменений, т.к. компилятор обеспечивает помещение внутрь массива `myShapes` только совместимых с `Shape` типов.

Сокрытие членов

Язык C# предоставляет возможность, которая логически противоположна переопределению методов и называется *сокрытием*. Выражаясь формально, если производный класс определяет член, который идентичен члену, определенному в базовом классе, то производный класс *скрывает* версию члена из родительского класса. В реальном мире такая ситуация чаще всего возникает, когда вы создаете подкласс от класса, который разрабатывали не вы (или ваша команда); например, такой класс может входить в состав программного пакета, приобретенного у стороннего поставщика.

В целях иллюстрации предположим, что вы получили от коллеги на доработку класс по имени `ThreeDCircle`, в котором определен метод `Draw()`, не принимающий аргументов:

```
class ThreeDCircle
{
  public void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

Вы полагаете, что `ThreeDCircle` "является" `Circle`, поэтому решаете унаследовать его от своего существующего типа `Circle`:

```
class ThreeDCircle : Circle
{
  public void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

После перекомпиляции вы обнаружите следующее предупреждение:

```
'ThreeDCircle.Draw()' hides inherited member 'Circle.Draw()'. To make
the current member override that implementation, add the override keyword.
Otherwise add the new keyword.
```

```
Shapes.ThreeDCircle.Draw() скрывает унаследованный член
Shapes.Circle.Draw(). Чтобы текущий член переопределял эту
реализацию, добавьте ключевое слово override.
В противном случае добавьте ключевое слово new.
```

Дело в том, что у вас есть производный класс (`ThreeDCircle`), который содержит метод, идентичный унаследованному методу. Решить проблему можно несколькими способами. Вы могли бы просто модифицировать версию метода `Draw()` в дочернем классе, добавив ключевое слово `override` (как предлагает компилятор). При таком подходе у типа `ThreeDCircle` появляется возможность расширять стандартное поведение родительского типа, как и требовалось. Однако если у вас нет доступа к файлу кода с определением базового класса (частый случай, когда приходится работать с множеством библиотек от сторонних поставщиков), тогда нет и возможности изменить метод `Draw()`, превратив его в виртуальный член.

В качестве альтернативы вы можете добавить ключевое слово `new` к определению проблемного члена `Draw()` своего производного типа (`ThreeDCircle`). Поступая так, вы явно утверждаете, что реализация производного типа намеренно спроектирована для фактического игнорирования версии члена из родительского типа (в реальности это может оказаться полезным, если внешнее программное обеспечение каким-то образом конфликтует с вашим программным обеспечением).

```
// Этот класс расширяет Circle и скрывает унаследованный метод Draw().
class ThreeDCircle : Circle
{
  // Скрыть любую реализацию Draw(), находящуюся выше в иерархии.
  public new void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

Вы можете также применить ключевое слово new к любому члену типа, который унаследован от базового класса (полю, константе, статическому члену или свойству). Продолжая пример, предположим, что в классе ThreeDCircle необходимо скрыть унаследованное свойство PetName:

```
class ThreeDCircle : Circle
{
  // Скрыть свойство PetName, определенное выше в иерархии.
  public new string PetName { get; set; }

  // Скрыть любую реализацию Draw(), находящуюся выше в иерархии.
  public new void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

Наконец, имейте в виду, что вы по-прежнему можете обратиться к реализации скрытого члена из базового класса, используя явное приведение, как описано в следующем разделе. Вот пример:

```
// Здесь вызывается метод Draw(), определенный в классе ThreeDCircle.
ThreeDCircle o = new ThreeDCircle();
o.Draw();

// Здесь вызывается метод Draw(), определенный в родительском классе!
((Circle)o).Draw();
Console.ReadLine();
```

Правила приведения для базовых и производных классов

Теперь, когда вы умеете строить семейства взаимосвязанных типов классов, нужно изучить правила, которым подчиняются *операции приведения классов*. Давайте возвратимся к иерархии классов для сотрудников, созданной ранее в главе, и добавим несколько новых методов в класс Program (если вы прорабатываете примеры, тогда откройте проект Employees в Visual Studio). Как описано в последнем разделе настоящей главы, изначальным базовым классом в системе является System.Object. По указанной причине любой класс "является" Object и может трактоваться как таковой. Таким образом, внутри переменной типа object допускается хранить экземпляр любого типа:

```
static void CastingExamples()
{
  // Manager "является" System.Object, поэтому в переменной
  // типа object можно сохранять ссылку на Manager.
  object frank = new Manager("Frank Zappa", 9, 3000, 40000,"111-11-1111", 5);
}
```

В проекте Employees классы Manager, SalesPerson и PtSalesPerson расширяют класс Employee, а потому допустимая ссылка на базовый класс может хранить любой из объектов указанных классов. Следовательно, приведенный далее код также законен:

```
static void CastingExamples()
{
  // Manager "является" System.Object, поэтому в переменной
  // типа object можно сохранять ссылку на Manager.
  object frank = new Manager("Frank Zappa", 9, 3000, 40000,
                             "111-11-1111", 5);

  // Manager тоже "является" Employee.
  Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000,
                                  "101-11-1321", 1);

  // PtSalesPerson "является" SalesPerson.
  SalesPerson jill = new PtSalesPerson("Jill", 834, 3002, 100000,
                                       "111-12-1119", 90);
}
```

Первое правило приведения между типами классов гласит, что когда два класса связаны отношением "является", то всегда можно безопасно сохранить объект производного типа в ссылке базового класса. Формально это называется *неявным приведением*, поскольку оно "просто работает" в соответствии с законами наследования. В результате появляется возможность строить некоторые мощные программные конструкции. Например, предположим, что в текущем классе `Program` определен новый метод:

```
static void GivePromotion(Employee emp)
{
  // Повысить зарплату...
  // Предоставить место на парковке компании...

  Console.WriteLine("{0} was promoted!", emp.Name);
}
```

Из-за того, что данный метод принимает единственный параметр типа `Employee`, в сущности, ему можно передавать объект любого унаследованного от `Employee` класса, учитывая наличие отношения "является":

```
static void CastingExamples()
{
  // Manager "является" System.Object, поэтому в переменной
  // типа object можно сохранять ссылку на Manager.
  object frank = new Manager("Frank Zappa", 9, 3000, 40000,"111-11-1111", 5);

  // Manager также "является" Employee.
  Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000,
                                  "101-11-1321", 1);
  GivePromotion(moonUnit);

  // PtSalesPerson "является" SalesPerson.
  SalesPerson jill = new PtSalesPerson("Jill", 834, 3002, 100000,
                                       "111-12-1119", 90);
  GivePromotion(jill);
}
```

Предыдущий код компилируется благодаря неявному приведению от типа базового класса (`Employee`) к производному классу. Но что, если вы хотите также вызвать метод `GivePromotion()` с объектом `frank` (хранящимся в общей ссылке `System.Object`)? Если вы передадите объект `frank` методу `GivePromotion()` напрямую, то получите ошибку на этапе компиляции:

```
object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
// Ошибка!
GivePromotion(frank);
```

Проблема в том, что вы пытаетесь передать переменную, которая объявлена как принадлежащая не к типу `Employee`, а к более общему типу `System.Object`. Учитывая, что в цепочке наследования он находится выше, чем `Employee`, компилятор не разрешит неявное приведение, стараясь сохранить ваш код насколько возможно безопасным в отношении типов.

Несмотря на то что сами вы можете выяснить, что ссылка `object` указывает в памяти на объект совместимого с `Employee` класса, компилятор сделать подобное не в состоянии, поскольку это не будет известно вплоть до времени выполнения. Чтобы удовлетворить компилятор, понадобится применить *явное приведение*, которое и является вторым правилом: в таких случаях вы можете явно приводить "вниз", используя операцию приведения C#. Базовый шаблон, которому нужно следовать при выполнении явного приведения, выглядит так:

```
(класс_к_которому_нужно_привести) существующая_ссылка
```

Таким образом, чтобы передать переменную типа `object` методу `GivePromotion()`, потребуется написать следующий код:

```
// Правильно!
GivePromotion((Manager)frank);
```

Использование ключевого слова `as`

Имейте в виду, что явное приведение оценивается *во время выполнения*, а не на этапе компиляции. Ради иллюстрации предположим, что проект `Employees` содержит копию класса `Hexagon`, созданного ранее в главе. Для простоты вы можете добавить в текущий проект такой класс:

```
class Hexagon
{
  public void Draw()
  {
    Console.WriteLine("Drawing a hexagon!");
  }
}
```

Хотя приведение объекта сотрудника к объекту фигуры абсолютно лишено смысла, код вроде показанного ниже скомпилируется без ошибок:

```
// Привести объект frank к типу Hexagon невозможно,
// но этот код нормально скомпилируется!
object frank = new Manager();
Hexagon hex = (Hexagon)frank;
```

Тем не менее, вы получите ошибку времени выполнения, или более формально — *исключение времени выполнения*. В главе 7 будут рассматриваться подробности структурированной обработки исключений, а пока полезно отметить, что при явном приведении можно перехватывать возможные ошибки с применением ключевых слов `try` и `catch`:

```
// Перехват возможной ошибки приведения.
object frank = new Manager();
```

```
Hexagon hex;
try
{
  hex = (Hexagon)frank;
}
catch (InvalidCastException ex)
{
  Console.WriteLine(ex.Message);
}
```

Очевидно, что показанный пример надуман; в такой ситуации вас никогда не будет беспокоить приведение между указанными типами. Однако предположим, что есть массив элементов `System.Object`, среди которых лишь малая толика содержит объекты, совместимые с `Employee`. В этом случае первым делом желательно определить, совместим ли элемент массива с типом `Employee`, и если да, то лишь тогда выполнить приведение.

Для быстрого определения совместимости одного типа с другим во время выполнения в C# предусмотрено ключевое слово `as`. С помощью ключевого слова `as` можно определить совместимость, проверив возвращаемое значение на предмет `null`. Взгляните на следующий код:

```
// Использование ключевого слова as для проверки совместимости.
object[] things = new object[4];
things[0] = new Hexagon();
things[1] = false;
things[2] = new Manager();
things[3] = "Last thing";
foreach (object item in things)
{
  Hexagon h = item as Hexagon;
  if (h == null)
  {
    Console.WriteLine("Item is not a hexagon");  // item - не Hexagon
  }
  else
  {
    h.Draw();
  }
}
```

Здесь производится проход в цикле по всем элементам в массиве объектов и проверка каждого из них на совместимость с классом `Hexagon`. Метод `Draw()` вызывается, если (и только если) обнаруживается объект, совместимый с `Hexagon`. В противном случае выводится сообщение о том, что элемент несовместим.

Использование ключевого слова `is` (обновление в версиях 7.0, 9.0)

В дополнение к ключевому слову `as` язык C# предлагает ключевое слово `is`, предназначенное для определения совместимости типов двух элементов. Тем не менее, в отличие от ключевого слова `as`, если типы не совместимы, тогда ключевое слово `is` возвращает `false`, а не ссылку `null`. В текущий момент метод `GivePromotion()` спроектирован для приема любого возможного типа, производного от `Employee`. Взгляните на следующую его модификацию, в которой теперь осуществляется проверка, какой конкретно "тип сотрудника" был передан:

```
static void GivePromotion(Employee emp)
{
  Console.WriteLine("{0} was promoted!", emp.Name);
  if (emp is SalesPerson)
  {
    Console.WriteLine("{0} made {1} sale(s)!", emp.Name,
      ((SalesPerson)emp).SalesNumber);
    Console.WriteLine();
  }
  else if (emp is Manager)
  {
    Console.WriteLine("{0} had {1} stock options...", emp.Name,
      ((Manager)emp).StockOptions);
    Console.WriteLine();
  }
}
```

Здесь во время выполнения производится проверка с целью выяснения, на что именно в памяти указывает входная ссылка типа базового класса. После определения, принят ли объект типа `SalesPerson` или `Manager`, можно применить явное приведение, чтобы получить доступ к специализированным членам данного типа. Также обратите внимание, что помещать операции приведения внутрь конструкции `try/catch` не обязательно, т.к. внутри раздела `if`, выполнившего проверку условия, уже известно, что приведение безопасно.

Начиная с версии C# 7.0, с помощью ключевого слова `is` переменной можно также присваивать объект преобразованного типа, если приведение работает. Это позволяет сделать предыдущий метод более ясным, устраняя проблему "двойного приведения". В предшествующем примере первое приведение выполняется, когда производится проверка совпадения типов, и если она проходит успешно, то переменную придется приводить снова. Взгляните на следующее обновление предыдущего метода:

```
static void GivePromotion(Employee emp)
{
  Console.WriteLine("{0} was promoted!", emp.Name);
  // Если SalesPerson, тогда присвоить переменной s.
  if (emp is SalesPerson s)
  {
    Console.WriteLine("{0} made {1} sale(s)!", s.Name, s.SalesNumber);
    Console.WriteLine();
  }
  // Если Manager, тогда присвоить переменной m.
  else if (emp is Manager m)
  {
    Console.WriteLine("{0} had {1} stock options...",m.Name,m.StockOptions);
    Console.WriteLine();
  }
}
```

В версии C# 9.0 появились дополнительные возможности сопоставления с образцом (они были раскрыты в главе 3). Такое обновленное сопоставление с образцом можно использовать с ключевым словом `is`. Например, для проверки, что объект сотрудника не относится ни к классу `Manager`, ни к классу `SalesPerson`, применяйте следующий код:

```
if (emp is not Manager and not SalesPerson)
{
  Console.WriteLine("Unable to promote {0}. Wrong employee type",
                    emp.Name);
  Console.WriteLine();
}
```

Использование отбрасывания вместе с ключевым словом `is` (нововведение в версии 7.0)

Ключевое слово `is` также разрешено применять в сочетании с заполнителем для отбрасывания переменных. Вот как можно обеспечить перехват объектов всех типов в операторе `if` или `switch`:

```
if (obj is var _)
{
  // Делать что-то.
}
```

Такое условие будет давать совпадение с чем угодно, а потому следует уделять внимание порядку, в котором используется блок сравнения с отбрасыванием. Ниже показан обновленный метод `GivePromotion()`:

```
if (emp is SalesPerson s)
{
  Console.WriteLine("{0} made {1} sale(s)!", s.Name, s.SalesNumber);
  Console.WriteLine();
}
// Если Manager, тогда присвоить переменной m.
else if (emp is Manager m)
{
  Console.WriteLine("{0} had {1} stock options...",
                    m.Name, m.StockOptions);
  Console.WriteLine();
}
else if (emp is var _)
{
  // Некорректный тип сотрудника.
  Console.WriteLine("Unable to promote {0}. Wrong employee type",
                    emp.Name);
  Console.WriteLine();
}
```

Финальный оператор `if` будет перехватывать любой экземпляр `Employee`, не являющийся `Manager`, `SalesPerson` или `PtSalesPerson`. Не забывайте, что вы можете приводить вниз к базовому классу, поэтому `PtSalesPerson` будет регистрироваться как `SalesPerson`.

Еще раз о сопоставлении с образцом (нововведение в версии 7.0)

В главе 3 было представлено средство сопоставления с образцом C# 7.0 наряду с его обновлениями в версии C# 9.0. Теперь, когда вы обрели прочное понимание приведения, наступило время для более удачного примера. Предыдущий пример можно модернизировать для применения оператора `switch`, сопоставляющего с образцом:

```
static void GivePromotion(Employee emp)
{
  Console.WriteLine("{0} was promoted!", emp.Name);
  switch (emp)
  {
    case SalesPerson s:
      Console.WriteLine("{0} made {1} sale(s)!", emp.Name, s.SalesNumber);
      break;
    case Manager m:
      Console.WriteLine("{0} had {1} stock options...",
                        emp.Name, m.StockOptions);
      break;
  }
  Console.WriteLine();
}
```

Когда к оператору `case` добавляется конструкция `when`, для использования доступно полное определение объекта *как он приводится*. Например, свойство `SalesNumber` существует только в классе `SalesPerson`, но не в классе `Employee`. Если приведение в первом операторе `case` проходит успешно, то переменная `s` будет содержать экземпляр класса `SalesPerson`, так что оператор `case` можно было бы переписать следующим образом:

```
case SalesPerson s when s.SalesNumber > 5:
```

Такие новые добавления к `is` и `switch` обеспечивают удобные улучшения, которые помогают сократить объем кода, выполняющего сопоставление, как демонстрировалось в предшествующих примерах.

Использование отбрасывания вместе с операторами `switch` (нововведение в версии 7.0)

Отбрасывание также может применяться в операторах `switch`:

```
switch (emp)
{
  case SalesPerson s when s.SalesNumber > 5:
    Console.WriteLine("{0} made {1} sale(s)!", emp.Name,
      s.SalesNumber);
    break;
  case Manager m:
    Console.WriteLine("{0} had {1} stock options...",
      emp.Name, m.StockOptions);
    break;
  case Employee _:
    // Некорректный тип сотрудника
    Console.WriteLine("Unable to promote{0}. Wrong employee type",emp.Name);
    break;
}
```

Каждый входной тип уже является `Employee` и потому финальный оператор `case` всегда дает `true`. Однако, как было показано при представлении сопоставления с образцом в главе 3, после сопоставления оператор `switch` завершает работу. Это демонстрирует важность правильности порядка. Если финальный оператор `case` переместить в начало, тогда никто из сотрудников не получит повышения.

Главный родительский класс: System.Object

В заключение мы займемся исследованием главного родительского класса — Object. При чтении предыдущих разделов вы могли заметить, что базовые классы во всех иерархиях (Car, Shape, Employee) никогда явно не указывали свои родительские классы:

```
// Какой класс является родительским для Car?
class Car
{...}
```

В мире .NET Core каждый тип в конечном итоге является производным от базового класса по имени System.Object, который в языке C# может быть представлен с помощью ключевого слова object (с буквой o в нижнем регистре). Класс Object определяет набор общих членов для каждого типа внутри платформы. По сути, когда вы строите класс, в котором явно не указан родительский класс, компилятор автоматически делает его производным от Object. Если вы хотите прояснить свои намерения, то можете определять классы, производные от Object, следующим образом (однако вы не обязаны поступать так):

```
// Явное наследование класса от System.Object.
class Car : object
{...}
```

Подобно любому классу в System.Object определен набор членов. В показанном ниже формальном определении C# обратите внимание, что некоторые члены объявлены как virtual, указывая на возможность их переопределения в подклассах, тогда как другие помечены ключевым словом static (и потому вызываются на уровне класса):

```
public class Object
{
  // Виртуальные члены.
  public virtual bool Equals(object obj);
  protected virtual void Finalize();
  public virtual int GetHashCode();
  public virtual string ToString();

  // Невиртуальные члены уровня экземпляра.
  public Type GetType();
  protected object MemberwiseClone();

  // Статические члены.
  public static bool Equals(object objA, object objB);
  public static bool ReferenceEquals(object objA, object objB);
}
```

В табл. 6.1 приведен обзор функциональности, предоставляемой некоторыми часто используемыми методами System.Object.

Чтобы проиллюстрировать стандартное поведение, обеспечиваемое базовым классом Object, создайте новый проект консольного приложения C# по имени ObjectOverrides.

Таблица 6.1. Основные методы `System.Object`

Метод экземпляра	Описание
`Equals()`	По умолчанию этот метод возвращает `true`, только если сравниваемые элементы ссылаются на один и тот же объект в памяти. Таким образом, `Equals()` применяется для сравнения объектных ссылок, а не состояния объектов. Обычно данный метод переопределяется, чтобы возвращать `true`, когда сравниваемые объекты имеют одинаковые значения внутреннего состояния (т.е. семантика, основанная на значениях). Имейте в виду, что если вы переопределяете `Equals()`, тогда должны также переопределить `GetHashCode()`, т.к. данные методы используются внутренне типами `Hashtable` для извлечения подобъектов из контейнера. Также вспомните из главы 4, что в классе `ValueType` этот метод переопределен для всех структур, чтобы выполнять сравнение на основе значений
`Finalize()`	Пока можно считать, что этот метод (когда он переопределен) вызывается для освобождения любых выделенных ресурсов перед уничтожением объекта. Сборка мусора в среде CoreCLR подробно рассматривается в главе 9
`GetHashCode()`	Этот метод возвращает значение `int`, которое идентифицирует конкретный объект
`ToString()`	Этот метод возвращает строковое представление объекта в формате `<пространство_имен>.<имя_типа>` (называемое *полностью заданным именем*). Он будет часто переопределяться в подклассе, чтобы вместо полностью заданного имени возвращать строку, которая состоит из пар "имя-значение", представляющих внутреннее состояние объекта
`GetType()`	Этот метод возвращает объект `Type`, полностью описывающий объект, на который в текущий момент производится ссылка. Другими словами, он является методом идентификации типов во время выполнения (runtime type identification — RTTI), доступным всем объектам (подробно обсуждается в главе 16)
`MemberwiseClone()`	Этот метод возвращает почленную копию текущего объекта и часто применяется для клонирования объектов (см. главу 8)

Добавьте в проект новый файл класса C#, содержащий следующее пустое определение типа `Person`:

```
// Не забывайте, что класс Person расширяет Object.
class Person {}
```

Теперь обновите операторы верхнего уровня для взаимодействия с унаследованными членами `System.Object`:

```
Console.WriteLine("***** Fun with System.Object *****\n");
Person p1 = new Person();

// Использовать унаследованные члены System.Object.
Console.WriteLine("ToString: {0}", p1.ToString());
Console.WriteLine("Hash code: {0}", p1.GetHashCode());
```

```
Console.WriteLine("Type: {0}", p1.GetType());

// Создать другие ссылки на p1.
Person p2 = p1;
object o = p2;

// Указывают ли ссылки на один и тот же объект в памяти?
if (o.Equals(p1) && p2.Equals(o))
{
  Console.WriteLine("Same instance!");
}
Console.ReadLine();
```

Вот вывод, получаемый в результате выполнения этого кода:

```
***** Fun with System.Object *****

ToString: ObjectOverrides.Person
Hash code: 58225482
Type: ObjectOverrides.Person
Same instance!
```

Обратите внимание на то, что стандартная реализация `ToString()` возвращает полностью заданное имя текущего типа (`ObjectOverrides.Person`). Как будет показано в главе 15, где исследуется построение специальных пространств имен, каждый проект C# определяет "корневое пространство имен", название которого совпадает с именем проекта. Здесь мы создали проект по имени `ObjectOverrides`, поэтому тип `Person` и класс `Program` помещены внутрь пространства имен `ObjectOverrides`.

Стандартное поведение метода `Equals()` заключается в проверке, указывают ли две переменные на один и тот же объект в памяти. В коде мы создаем новую переменную `Person` по имени `p1`. В этот момент новый объект `Person` помещается в управляемую кучу. Переменная `p2` также относится к типу `Person`. Тем не менее, вместо создания *нового* экземпляра переменной `p2` присваивается ссылка `p1`. Таким образом, переменные `p1` и `p2` указывают на один и тот же объект в памяти, как и переменная `o` (типа `object`). Учитывая, что `p1`, `p2` и `o` указывают на одно и то же местоположение в памяти, проверка эквивалентности дает положительный результат.

Хотя готовое поведение `System.Object` в ряде случаев может удовлетворять всем потребностям, довольно часто в специальных типах часть этих унаследованных методов переопределяется. В целях иллюстрации модифицируем класс `Person`, добавив свойства, которые представляют имя, фамилию и возраст лица; все они могут быть установлены с помощью специального конструктора:

```
// Не забывайте, что класс Person расширяет Object.
class Person
{
  public string FirstName { get; set; } = "";
  public string LastName { get; set; } = "";
  public int Age { get; set; }
  public Person(string fName, string lName, int personAge)
  {
    FirstName = fName;
    LastName = lName;
    Age = personAge;
  }
  public Person(){}
}
```

Переопределение метода System.Object.ToString()

Многие создаваемые классы (и структуры) могут извлечь преимущества от переопределения метода ToString() для возвращения строки с текстовым представлением текущего состояния экземпляра типа. Помимо прочего это полезно при отладке. То, как вы решите конструировать результирующую строку — дело личных предпочтений; однако рекомендуемый подход предусматривает отделение пар "имя-значение" друг от друга двоеточиями и помещение всей строки в квадратные скобки (такому принципу следуют многие типы из библиотек базовых классов .NET Core). Взгляните на следующую переопределенную версию ToString() для класса Person:

```
public override string ToString()
  => $"[First Name: {FirstName}; Last Name: {LastName}; Age: {Age}]";
```

Приведенная реализация метода ToString() довольно прямолинейна, потому что класс Person содержит всего три порции данных состояния. Тем не менее, всегда помните о том, что правильное переопределение ToString() должно также учитывать любые данные, определенные *выше в цепочке наследования*.

При переопределении метода ToString() для класса, расширяющего специальный базовый класс, первым делом необходимо получить возвращаемое значение ToString() из родительского класса, используя ключевое слово base. После получения строковых данных родительского класса их можно дополнить специальной информацией производного класса.

Переопределение метода System.Object.Equals()

Давайте также переопределим поведение метода Object.Equals(), чтобы работать с *семантикой на основе значений*. Вспомните, что по умолчанию Equals() возвращает true, только если два сравниваемых объекта ссылаются на один и тот же экземпляр объекта в памяти. Для класса Person может оказаться полезной такая реализация Equals(), которая возвращает true, если две сравниваемые переменные содержат те же самые значения состояния (например, фамилию, имя и возраст).

Прежде всего, обратите внимание, что входной аргумент метода Equals() имеет общий тип System.Object. В связи с этим первым делом необходимо удостовериться в том, что вызывающий код действительно передал экземпляр типа Person, и для дополнительной подстраховки проверить, что входной параметр не является ссылкой null.

После того, как вы установите, что вызывающий код передал выделенный экземпляр Person, один из подходов предусматривает реализацию метода Equals() для сравнения поле за полем данных входного объекта с данными текущего объекта:

```
public override bool Equals(object obj)
{
  if (!(obj is Person temp))
  {
    return false;
  }
  if (temp.FirstName == this.FirstName
      && temp.LastName == this.LastName
      && temp.Age == this.Age)
  {
    return true;
  }
  return false;
}
```

Здесь производится сравнение значений входного объекта с внутренними значениями текущего объекта (обратите внимание на применение ключевого слова this). Если имя, фамилия и возраст в двух объектах идентичны, то эти два объекта имеют одинаковые данные состояния и возвращается значение true. Любые другие результаты приводят к возвращению false.

Хотя такой подход действительно работает, вы определенно в состоянии представить, насколько трудоемкой была бы реализация специального метода Equals() для нетривиальных типов, которые могут содержать десятки полей данных. Распространенное сокращение предусматривает использование собственной реализации метода ToString(). Если класс располагает подходящей реализацией ToString(), в которой учитываются все поля данных вверх по цепочке наследования, тогда можно просто сравнивать строковые данные объектов (проверив на равенство null):

```
// Больше нет необходимости приводить obj к типу Person,
// т.к. у всех типов имеется метод ToString().
public override bool Equals(object obj)
  => obj?.ToString() == ToString();
```

Обратите внимание, что в этом случае нет необходимости проверять входной аргумент на принадлежность к корректному типу (Person в нашем примере), поскольку метод ToString() поддерживают все типы .NET. Еще лучше то, что больше не требуется выполнять проверку на предмет равенства свойство за свойством, т.к. теперь просто проверяются значения, возвращаемые методом ToString().

Переопределение метода System.Object.GetHashCode()

В случае переопределения в классе метода Equals() вы также должны переопределить стандартную реализацию метода GetHashCode(). Выражаясь упрощенно, *хеш-код* — это числовое значение, которое представляет объект как специфическое состояние. Например, если вы создадите две переменные типа string, хранящие значение Hello, то они должны давать один и тот же хеш-код. Однако если одна из них хранит строку в нижнем регистре (hello), то должны получаться разные хеш-коды.

Для выдачи хеш-значения метод System.Object.GetHashCode() по умолчанию применяет адрес текущей ячейки памяти, где расположен объект. Тем не менее, если вы строите специальный тип, подлежащий хранению в экземпляре типа Hashtable (из пространства имен System.Collections), тогда всегда должны переопределять данный член, потому что для извлечения объекта тип Hashtable будет вызывать методы Equals() и GetHashCode().

На заметку! Говоря точнее, класс System.Collections.Hashtable внутренне вызывает метод GetHashCode(), чтобы получить общее представление о местоположении объекта, а с помощью последующего (внутреннего) вызова метода Equals() определяет его точно.

Хотя в настоящем примере мы не собираемся помещать объекты Person внутрь System.Collections.Hashtable, ради полноты изложения давайте переопределим метод GetHashCode(). Существует много алгоритмов, которые можно применять для создания хеш-кода, как весьма изощренных, так и не очень. В большинстве ситуаций есть возможность генерировать значение хеш-кода, полагаясь на реализацию метода GetHashCode() из класса System.String.

Учитывая, что класс `String` уже имеет эффективный алгоритм хеширования, использующий для вычисления хеш-значения символьные данные объекта `String`, вы можете просто вызвать метод `GetHashCode()` с той частью полей данных, которая должна быть уникальной во всех экземплярах (вроде номера карточки социального страхования), если ее удается идентифицировать. Таким образом, если в классе `Person` определено свойство `SSN`, то вы могли бы написать следующий код:

```
// Предположим, что имеется свойство SSN.
class Person
{
  public string SSN { get; } = "";
  public Person(string fName, string lName, int personAge,
    string ssn)
  {
    FirstName = fName;
    LastName = lName;
    Age = personAge;
    SSN = ssn;
  }
  // Возвратить хеш-код на основе уникальных строковых данных.
  public override int GetHashCode() => SSN.GetHashCode();
}
```

В случае использования в качестве основы хеш-кода свойства, допускающего чтение и запись, вы получите предупреждение. После того, как объект создан, хеш-код должен быть неизменяемым. В предыдущем примере свойство `SSN` имеет только метод `get`, что делает его допускающим только чтение, и устанавливать его можно только в конструкторе.

Если вы не можете отыскать единый фрагмент уникальных строковых данных, но есть переопределенный метод `ToString()`, который удовлетворяет соглашению о доступе только по чтению, тогда вызывайте `GetHashCode()` на собственном строковом представлении:

```
// Возвратить хеш-код на основе значения, возвращаемого
// методом ToString() для объекта Person.
public override int GetHashCode()
{
  return this.ToString().GetHashCode();
}
```

Тестирование модифицированного класса `Person`

Теперь, когда виртуальные члены класса `Object` переопределены, обновите операторы верхнего уровня, чтобы протестировать внесенные изменения:

```
Console.WriteLine("***** Fun with System.Object *****\n");

// ПРИМЕЧАНИЕ: мы хотим, чтобы эти объекты были идентичными
// в целях тестирования методов Equals() и GetHashCode().
Person p1 = new Person("Homer", "Simpson", 50, "111-11-1111");
Person p2 = new Person("Homer", "Simpson", 50, "111-11-1111");

// Получить строковые версии объектов.
Console.WriteLine("p1.ToString() = {0}", p1.ToString());
Console.WriteLine("p2.ToString() = {0}", p2.ToString());
```

```
// Протестировать переопределенный метод Equals().
Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));

// Протестировать хеш-коды.
// По-прежнему используется хеш-значение SSN.
Console.WriteLine("Same hash codes?: {0}",
                  p1.GetHashCode() == p2.GetHashCode());
Console.WriteLine();

// Изменить значение Age объекта p2 и протестировать снова.
p2.Age = 45;
Console.WriteLine("p1.ToString() = {0}", p1.ToString());
Console.WriteLine("p2.ToString() = {0}", p2.ToString());
Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));

// По-прежнему используется хеш-значение SSN.
Console.WriteLine("Same hash codes?: {0}",
                  p1.GetHashCode() == p2.GetHashCode());
Console.ReadLine();
```

Ниже показан вывод:

```
***** Fun with System.Object *****
p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p1 = p2?: True
Same hash codes?: True

p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 45]
p1 = p2?: False
Same hash codes?: True
```

Использование статических членов класса `System.Object`

В дополнение к только что рассмотренным членам уровня экземпляра класс `System.Object` определяет два статических члена, которые также проверяют эквивалентность на основе значений или на основе ссылок. Взгляните на следующий код:

```
static void StaticMembersOfObject()
{
  // Статические члены System.Object.
  Person p3 = new Person("Sally", "Jones", 4);
  Person p4 = new Person("Sally", "Jones", 4);
  Console.WriteLine("P3 and P4 have same state: {0}",
                    object.Equals(p3, p4));
                  // P3 и P4 имеют то же самое состояние
  Console.WriteLine("P3 and P4 are pointing to same object: {0}",
                    object.ReferenceEquals(p3, p4));
                  // P3 и P4 указывают на тот же самый объект
}
```

Здесь вы имеете возможность просто отправить два объекта (любого типа) и позволить классу `System.Object` выяснить детали автоматически.

Ниже показан вывод, полученный в результате вызова метода `StaticMembersOfObject()` в операторах верхнего уровня:

```
***** Fun with System.Object *****
P3 and P4 have the same state: True
P3 and P4 are pointing to the same object: False
```

Резюме

В настоящей главе объяснялась роль и детали наследования и полиморфизма. В ней были представлены многочисленные новые ключевые слова и лексемы для поддержки каждого приема. Например, вспомните, что с помощью двоеточия указывается родительский класс для создаваемого типа. Родительские типы способны определять любое количество виртуальных и/или абстрактных членов для установления полиморфного интерфейса. Производные типы переопределяют эти члены с применением ключевого слова `override`.

Вдобавок к построению множества иерархий классов в главе также исследовалось явное приведение между базовыми и производными типами. В завершение главы рассматривались особенности главного родительского класса в библиотеках базовых классов .NET Core — `System.Object`.

ГЛАВА 7

Структурированная обработка исключений

В настоящей главе вы узнаете о том, как иметь дело с аномалиями, возникающими во время выполнения кода C#, с использованием *структурированной обработки исключений*. Будут описаны не только ключевые слова C#, предназначенные для этих целей (`try`, `catch`, `throw`, `finally`, `when`), но и разница между исключениями уровня приложения и уровня системы, а также роль базового класса `System.Exception`. Кроме того, будет показано, как создавать специальные исключения, и рассмотрены некоторые инструменты отладки в Visual Studio, связанные с исключениями.

Ода ошибкам, дефектам и исключениям

Что бы ни нашептывало наше (порой завышенное) самомнение, идеальных программистов не существует. Разработка программного обеспечения является сложным делом, и из-за такой сложности довольно часто даже самые лучшие программы поставляются с разнообразными *проблемами*. В одних случаях проблема возникает из-за "плохо написанного" кода (например, по причине выхода за границы массива), а в других — из-за ввода пользователем некорректных данных, которые не были учтены в кодовой базе приложения (скажем, когда в поле для телефонного номера вводится значение вроде `Chucky`). Вне зависимости от причин проблемы в конечном итоге приложение не работает ожидаемым образом. Чтобы подготовить почву для предстоящего обсуждения структурированной обработки исключений, рассмотрим три распространенных термина, которые применяются для описания аномалий.

- **Дефекты.** Выражаясь просто, это ошибки, которые допустил программист. В качестве примера предположим, что вы программируете на неуправляемом C++. Если вы забудете освободить динамически выделенную память, что приводит к утечке памяти, тогда получите дефект.
- **Пользовательские ошибки.** С другой стороны, пользовательские ошибки обычно возникают из-за тех, кто запускает приложение, а не тех, кто его создает. Например, ввод конечным пользователем в текстовом поле неправильно сформированной строки с высокой вероятностью может привести к генерации ошибки, если в коде не была предусмотрена обработка некорректного ввода.
- **Исключения.** Исключениями обычно считаются аномалии во время выполнения, которые трудно (а то и невозможно) учесть на стадии программирования приложения. Примерами исключений могут быть попытка подключения к базе

данных, которая больше не существует, открытие запорченного XML-файла или попытка установления связи с машиной, которая в текущий момент находится в автономном режиме. В каждом из упомянутых случаев программист (или конечный пользователь) обладает довольно низким контролем над такими "исключительными" обстоятельствами.

С учетом приведенных определений должно быть понятно, что структурированная обработка *исключений* в .NET — прием работы с *исключительными ситуациями* во время выполнения. Тем не менее, даже для дефектов и пользовательских ошибок, которые ускользнули от глаз программиста, исполняющая среда будет часто генерировать соответствующее исключение, идентифицирующее возникшую проблему. Скажем, в библиотеках базовых классов .NET 5 определены многочисленные исключения, такие как `FormatException`, `IndexOutOfRangeException`, `FileNotFoundException`, `ArgumentOutOfRangeException` и т.д.

В рамках терминологии .NET *исключение* объясняется дефектами, некорректным пользовательским вводом и ошибками времени выполнения, даже если программисты могут трактовать каждую аномалию как отдельную проблему. Однако прежде чем погружаться в детали, формализуем роль структурированной обработки исключений и посмотрим, чем она отличается от традиционных приемов обработки ошибок.

На заметку! Чтобы сделать примеры кода максимально ясными, мы не будем перехватывать абсолютно все исключения, которые может выдавать заданный метод из библиотеки базовых классов. Разумеется, в своих проектах производственного уровня вы должны широко использовать приемы, описанные в главе.

Роль обработки исключений .NET

До появления платформы .NET обработка ошибок в среде операционной системы Windows представляла собой запутанную смесь технологий. Многие программисты внедряли собственную логику обработки ошибок в контекст разрабатываемого приложения. Например, команда разработчиков могла определять набор числовых констант для представления известных условий возникновения ошибок и затем применять эти константы как возвращаемые значения методов. Взгляните на следующий фрагмент кода на языке C:

```
/* Типичный механизм перехвата ошибок в стиле C. */
#define E_FILENOTFOUND 1000

int UseFileSystem()
{
  // Предполагается, что в этой функции происходит нечто
  // такое, что приводит к возврату следующего значения.
  return E_FILENOTFOUND;
}

void main()
{
  int retVal = UseFileSystem();
  if(retVal == E_FILENOTFOUND)
    printf("Cannot find file...");  // Не удалось найти файл
}
```

Такой подход далек от идеала, учитывая тот факт, что константа `E_FILENOTFOUND` — всего лишь числовое значение, которое немногое говорит о том, каким образом решить возникшую проблему. В идеале желательно, чтобы название ошибки, описательное сообщение и другая полезная информация об условиях возникновения ошибки были помещены в единственный четко определенный пакет (что как раз и происходит при структурированной обработке исключений). В дополнение к специальным приемам, к которым прибегают разработчики, внутри API-интерфейса Windows определены сотни кодов ошибок, которые поступают в виде определений `#define` и `HRESULT`, а также очень многих вариаций простых булевских значений (`bool`, `BOOL`, `VARIANT_BOOL` и т.д.).

Очевидной проблемой, присущей таким старым приемам, является полное отсутствие симметрии. Каждый подход более или менее подгоняется под заданную технологию, заданный язык и возможно даже заданный проект. Чтобы положить конец такому безумству, платформа .NET предложила стандартную методику для генерации и перехвата ошибок времени выполнения — структурированную обработку исключений. Достоинство этой методики в том, что разработчики теперь имеют унифицированный подход к обработке ошибок, который является общим для всех языков, ориентированных на .NET. Следовательно, способ обработки ошибок, используемый программистом на C#, синтаксически подобен способу, который применяет программист на VB или программист на C++, имеющий дело с C++/CLI.

Дополнительное преимущество связано с тем, что синтаксис, используемый для генерации и отлавливания исключений за пределами границ сборок и машины, идентичен. Скажем, если вы применяете язык C# при построении REST-службы ASP.NET Core, то можете сгенерировать исключение JSON для удаленного вызывающего кода, используя те же самые ключевые слова, которые позволяют генерировать исключения внутри методов одного приложения.

Еще одно преимущество исключений .NET состоит в том, что в отличие от загадочных числовых значений они представляют собой объекты, в которых содержится читабельное описание проблемы, а также детальный снимок стека вызовов на момент первоначального возникновения исключения. Более того, конечному пользователю можно предоставить справочную ссылку, которая указывает на URL-адрес с подробностями об ошибке, а также специальные данные, определенные программистом.

Строительные блоки обработки исключений в .NET

Программирование со структурированной обработкой исключений предусматривает применение четырех взаимосвязанных сущностей:

- тип класса, который представляет детали исключения;
- член, способный генерировать экземпляр класса исключения в вызывающем коде при соответствующих обстоятельствах;
- блок кода на вызывающей стороне, который обращается к члену, предрасположенному к возникновению исключения;
- блок кода на вызывающей стороне, который будет обрабатывать (или перехватывать) исключение, если оно возникнет.

Язык программирования C# предлагает пять ключевых слов (`try`, `catch`, `throw`, `finally` и `when`), которые позволяют генерировать и обрабатывать исключения. Объект, представляющий текущую проблему, относится к классу, который расширяет класс `System.Exception` (или производный от него класс). С учетом сказанного давайте исследуем роль данного базового класса, касающегося исключений.

Базовый класс System.Exception

Все исключения в конечном итоге происходят от базового класса System.Exception, который в свою очередь является производным от System.Object. Ниже показана основная часть этого класса (обратите внимание, что некоторые его члены являются виртуальными и, следовательно, могут быть переопределены в производных классах):

```
public class Exception : ISerializable
{
  // Открытые конструкторы.
  public Exception(string message, Exception innerException);
  public Exception(string message);
  public Exception();
  ...
  // Методы.
  public virtual Exception GetBaseException();
  public virtual void GetObjectData(SerializationInfo info,
    StreamingContext context);
  // Свойства.
  public virtual IDictionary Data { get; }
  public virtual string HelpLink { get; set; }
  public int HResult { get; set; }
  public Exception InnerException { get; }
  public virtual string Message { get; }
  public virtual string Source { get; set; }
  public virtual string StackTrace { get; }
  public MethodBase TargetSite { get; }
}
```

Как видите, многие свойства, определенные в классе System.Exception, по своей природе допускают только чтение. Причина в том, что стандартные значения для каждого из них обычно будут предоставляться производными типами. Например, стандартное сообщение типа IndexOutOfRangeException выглядит так: "Index was outside the bounds of the array" (Индекс вышел за границы массива).

В табл. 7.1 описаны наиболее важные члены класса System.Exception.

Таблица 7.1. Основные члены типа System.Exception

Свойство System.Exception	Описание
Data	Это свойство только для чтения позволяет извлекать коллекцию пар "ключ-значение" (представленную объектом, реализующим IDictionary), которая предоставляет дополнительную определяемую программистом информацию об исключении. По умолчанию коллекция пуста
HelpLink	Это свойство позволяет получать или устанавливать URL для доступа к справочному файлу или веб-сайту с подробным описанием ошибки
InnerException	Это свойство только для чтения может использоваться для получения информации о предыдущих исключениях, которые послужили причиной возникновения текущего исключения. Запись предыдущих исключений осуществляется путем их передачи конструктору самого последнего сгенерированного исключения

Окончание табл. 7.1

Свойство System.Exception	Описание
Message	Это свойство только для чтения возвращает текстовое описание заданной ошибки. Само сообщение об ошибке устанавливается как параметр конструктора
Source	Это свойство позволяет получать либо устанавливать имя сборки или объекта, который привел к генерации исключения
StackTrace	Это свойство только для чтения содержит строку, идентифицирующую последовательность вызовов, которая привела к возникновению исключения. Как нетрудно догадаться, данное свойство очень полезно во время отладки или для сохранения информации об ошибке во внешнем журнале ошибок
TargetSite	Это свойство только для чтения возвращает объект MethodBase с описанием многочисленных деталей о методе, который привел к генерации исключения (вызов ToString() будет идентифицировать этот метод по имени)

Простейший пример

Чтобы продемонстрировать полезность структурированной обработки исключений, мы должны создать класс, который будет генерировать исключение в надлежащих (или, можно сказать, *исключительных*) обстоятельствах. Создадим новый проект консольного приложения C# по имени SimpleException и определим в нем два класса (Car (автомобиль) и Radio (радиоприемник)), связав их между собой отношением "имеет". В классе Radio определен единственный метод, который отвечает за включение и выключение радиоприемника:

```
using System;
namespace SimpleException
{
  class Radio
  {
    public void TurnOn(bool on)
    {
      Console.WriteLine(on ? "Jamming..." : "Quiet time...");
    }
  }
}
```

В дополнение к использованию класса Radio через включение/делегацию класс Car (его код показан ниже) определен так, что если пользователь превышает предопределенную максимальную скорость (заданную с помощью константного члена MaxSpeed), тогда двигатель выходит из строя, приводя объект Car в нерабочее состояние (отражается закрытой переменной-членом типа bool по имени _carIsDead).

Кроме того, класс Car имеет несколько свойств для представления текущей скорости и указанного пользователем "дружественного названия" автомобиля, а также различные конструкторы для установки состояния нового объекта Car. Ниже приведено полное определение Car вместе с поясняющими комментариями.

```
using System;

namespace SimpleException
{
  class Car
  {
    // Константа для представления максимальной скорости.
    public const int MaxSpeed = 100;

    // Свойства автомобиля.
    public int CurrentSpeed {get; set;} = 0;
    public string PetName {get; set;} = "";

    // Не вышел ли автомобиль из строя?
    private bool _carIsDead;

    // В автомобиле имеется радиоприемник.
    private readonly Radio _theMusicBox = new Radio();

    // Конструкторы.
    public Car() {}
    public Car(string name, int speed)
    {
      CurrentSpeed = speed;
      PetName = name;
    }

    public void CrankTunes(bool state)
    {
      // Делегировать запрос внутреннему объекту.
      _theMusicBox.TurnOn(state);
    }

    // Проверить, не перегрелся ли автомобиль.
    public void Accelerate(int delta)
    {
      if (_carIsDead)
      {
        Console.WriteLine("{0} is out of order...", PetName);
      }
      else
      {
        CurrentSpeed += delta;
        if (CurrentSpeed > MaxSpeed)
        {
          Console.WriteLine("{0} has overheated!", PetName);
          CurrentSpeed = 0;
          _carIsDead = true;
        }
        else
        {
          Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
        }
      }
    }
  }
}
```

Обновите код в файле Program.cs, чтобы заставить объект Car превышать заранее заданную максимальную скорость (установленную в 100 внутри класса Car):

```
using System;
using System.Collections;
using SimpleException;

Console.WriteLine("***** Simple Exception Example *****");
Console.WriteLine("=> Creating a car and stepping on it!");
Car myCar = new Car("Zippy", 20);
myCar.CrankTunes(true);

for (int i = 0; i < 10; i++)
{
  myCar.Accelerate(10);
}
Console.ReadLine();
```

В результате запуска кода будет получен следующий вывод:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100
Zippy has overheated!
Zippy is out of order...
```

Генерация общего исключения

Теперь, имея функциональный класс Car, давайте рассмотрим простейший способ генерации исключения. Текущая реализация метода Accelerate() просто отображает сообщение об ошибке, если вызывающий код пытается разогнать автомобиль до скорости, превышающей верхний предел.

Чтобы модернизировать метод Accelerate() для генерации исключения, когда пользователь пытается разогнать автомобиль до скорости, которая превышает установленный предел, потребуется создать и сконфигурировать новый экземпляр класса System.Exception, установив значение доступного только для чтения свойства Message через конструктор класса. Для отправки объекта ошибки обратно вызывающему коду применяется ключевое слово throw языка C#. Ниже приведен обновленный код метода Accelerate():

```
// На этот раз генерировать исключение, если пользователь
// превышает предел, указанный в MaxSpeed.
public void Accelerate(int delta)
{
  if (_carIsDead)
  {
    Console.WriteLine("{0} is out of order...", PetName);
  }
```

```
    else
    {
      CurrentSpeed += delta;
      if (CurrentSpeed >= MaxSpeed)
      {
        CurrentSpeed = 0;
        _carIsDead = true;
        // Использовать ключевое слово throw для генерации исключения.
        throw new Exception($"{PetName} has overheated!");
      }
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
    }
  }
```

Прежде чем выяснять, каким образом вызывающий код будет перехватывать данное исключение, необходимо отметить несколько интересных моментов. Для начала, если вы генерируете исключение, то всегда самостоятельно решаете, как вводится в действие ошибка и когда должно генерироваться исключение. Здесь мы предполагаем, что при попытке увеличить скорость объекта `Car` за пределы максимума должен быть сгенерирован объект `System.Exception` для уведомления о невозможности продолжить выполнение метода `Accelerate()` (в зависимости от создаваемого приложения такое предположение может быть как допустимым, так и нет).

В качестве альтернативы метод `Accelerate()` можно было бы реализовать так, чтобы он производил автоматическое восстановление, не генерируя перед этим исключение. По большому счету исключения должны генерироваться только при возникновении более критичного условия (например, отсутствие нужного файла, невозможность подключения к базе данных и т.п.) и не использоваться как механизм потока логики. Принятие решения о том, что должно служить причиной генерации исключения, требует серьезного обдумывания и поиска веских оснований на стадии проектирования. Для преследуемых сейчас целей будем считать, что попытка увеличить скорость автомобиля выше максимально допустимой является вполне оправданной причиной для выдачи исключения.

Кроме того, обратите внимание, что из кода метода был удален финальный оператор `else`. Когда исключение генерируется (либо инфраструктурой, либо вручную с применением оператора `throw`), управление возвращается вызывающему методу (или блоку `catch` в операторе `try`). Это устраняет необходимость в финальном `else`. Оставите вы его ради лучшей читабельности или нет, зависит от ваших стандартов написания кода.

В любом случае, если вы снова запустите приложение с показанной ранее логикой в операторах верхнего уровня, то исключение в итоге будет сгенерировано. В показанном далее выводе видно, что результат отсутствия обработки этой ошибки нельзя назвать идеальным, учитывая получение многословного сообщения об ошибке (с вашим путем к файлу и номерами строк) и последующее прекращение работы программы:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
```

```
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100

Unhandled exception. System.Exception: Zippy has overheated!
  at SimpleException.Car.Accelerate(Int32 delta)
  in [путь к файлу]\Car.cs:line 52
  at SimpleException.Program.Main(String[] args)
  in [путь к файлу]\Program.cs:line 16
```

Перехват исключений

На заметку! Те, кто пришел в .NET 5 из мира Java, должны помнить о том, что члены типа не прототипируются набором исключений, которые они могут генерировать (другими словами, платформа .NET Core не поддерживает проверяемые исключения). Лучше это или хуже, но вы не обязаны обрабатывать каждое исключение, генерируемое отдельно взятым членом.

Поскольку метод `Accelerate()` теперь генерирует исключение, вызывающий код должен быть готов обработать его, если оно возникнет. При вызове метода, который может сгенерировать исключение, должен использоваться блок `try/catch`. После перехвата объекта исключения можно обращаться к различным его членам и извлекать детальную информацию о проблеме.

Дальнейшие действия с такими данными в значительной степени зависят от вас. Вы можете зафиксировать их в файле отчета, записать в журнал событий, отправить по электронной почте системному администратору или отобразить конечному пользователю сообщение о проблеме. Здесь мы просто выводим детали исключения в окно консоли:

```
// Обработка сгенерированного исключения.
Console.WriteLine("***** Simple Exception Example *****");
Console.WriteLine("=> Creating a car and stepping on it!");
Car myCar = new Car("Zippy", 20);
myCar.CrankTunes(true);

// Разогнаться до скорости, превышающей максимальный
// предел автомобиля, с целью выдачи исключения.
try
{
  for(int i = 0; i < 10; i++)
  {
    myCar. Accelerate(10);
  }
}
catch(Exception e)
{
  Console.WriteLine("\n*** Error! ***");              // ошибка
  Console.WriteLine("Method: {0}", e.TargetSite);     // метод
  Console.WriteLine("Message: {0}", e.Message);       // сообщение
  Console.WriteLine("Source: {0}", e.Source);         // источник
}
```

```
// Ошибка была обработана, выполнение продолжается со следующего оператора.
Console.WriteLine("\n***** Out of exception logic *****");
Console.ReadLine();
```

По существу блок `try` представляет собой раздел операторов, которые в ходе выполнения могут генерировать исключение. Если исключение обнаруживается, тогда управление переходит к соответствующему блоку `catch`. С другой стороны, если код внутри блока `try` исключение не сгенерировал, то блок `catch` полностью пропускается, и выполнение проходит обычным образом. Ниже представлен вывод, полученный в результате тестового запуска данной программы:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100
*** Error! ***
Method: Void Accelerate(Int32)
Message: Zippy has overheated!
Source: SimpleException
***** Out of exception logic *****
```

Как видите, после обработки исключения приложение может продолжать свое функционирование с оператора, находящегося после блока `catch`. В некоторых обстоятельствах исключение может оказаться достаточно критическим для того, чтобы служить основанием завершения работы приложения. Тем не менее, во многих случаях логика внутри обработчика исключений позволяет приложению спокойно продолжить выполнение (хотя, может быть, с несколько меньшим объемом функциональности, например, без возможности взаимодействия с удаленным источником данных).

Выражение `throw` (нововведение в версии 7.0)

До выхода версии C# 7 ключевое слово `throw` было оператором, что означало возможность генерации исключения только там, где разрешены операторы. В C# 7.0 и последующих версиях ключевое слово `throw` доступно также в виде выражения и может использоваться везде, где разрешены выражения.

Конфигурирование состояния исключения

В настоящий момент объект `System.Exception`, сконфигурированный внутри метода `Accelerate()`, просто устанавливает значение, доступное через свойство `Message` (посредством параметра конструктора). Как было показано ранее в табл. 7.1, класс `Exception` также предлагает несколько дополнительных членов (`TargetSite`, `StackTrace`, `HelpLink` и `Data`), которые полезны для дальнейшего уточнения природы возникшей проблемы. Чтобы усовершенствовать текущий пример, давайте по очереди рассмотрим возможности упомянутых членов.

Свойство TargetSite

Свойство System.Exception.TargetSite позволяет выяснить разнообразные детали о методе, который сгенерировал заданное исключение. Как демонстрировалось в предыдущем примере кода, в результате вывода значения свойства TargetSite отобразится возвращаемое значение, имя и типы параметров метода, который сгенерировал исключение. Однако свойство TargetSite возвращает не простую строку, а строго типизированный объект System.Reflection.MethodBase. Данный тип можно применять для сбора многочисленных деталей, касающихся проблемного метода, а также класса, в котором метод определен. В целях иллюстрации измените предыдущую логику в блоке catch следующим образом:

```
...
// Свойство TargetSite в действительности возвращает объект MethodBase.
catch(Exception e)
{
  Console.WriteLine("\n*** Error! ***");
  Console.WriteLine("Member name: {0}", e.TargetSite);    // имя члена
  Console.WriteLine("Class defining member: {0}",
    e.TargetSite.DeclaringType);          // класс, определяющий член
  Console.WriteLine("Member type: {0}", e.TargetSite.MemberType);
                  // тип члена
  Console.WriteLine("Message: {0}", e.Message);           // сообщение
  Console.WriteLine("Source: {0}", e.Source);             // источник
}
Console.WriteLine("\n***** Out of exception logic *****");
Console.ReadLine();
```

На этот раз в коде используется свойство MethodBase.DeclaringType для выяснения полностью заданного имени класса, сгенерировавшего ошибку (в данном случае SimpleException.Car), а также свойство MemberType объекта MethodBase для идентификации вида члена (например, член является свойством или методом), в котором возникло исключение. Ниже показано, как будет выглядеть вывод в результате выполнения логики в блоке catch:

```
*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
```

Свойство StackTrace

Свойство System.Exception.StackTrace позволяет идентифицировать последовательность вызовов, которая в результате привела к генерации исключения. Значение данного свойства никогда не устанавливается вручную — это делается автоматически во время создания объекта исключения. Чтобы удостовериться в сказанном, модифицируйте логику в блоке catch:

```
catch(Exception e)
{
  ...
  Console.WriteLine("Stack: {0}", e.StackTrace);
}
```

Снова запустив программу, в окне консоли можно обнаружить следующие данные трассировки стека (естественно, номера строк и пути к файлам у вас могут отличаться):

```
Stack: at SimpleException.Car.Accelerate(Int32 delta)
in [путь к файлу]\car.cs:line 57 at <Program>$.<Main>$(String[] args)
in [путь к файлу]\Program.cs:line 20
```

Значение типа `string`, возвращаемое свойством `StackTrace`, отражает последовательность вызовов, которая привела к генерации данного исключения. Обратите внимание, что самый нижний номер строки в `string` указывает на место возникновения первого вызова в последовательности, а самый верхний — на место, где точно находится проблемный член. Очевидно, что такая информация очень полезна во время отладки или при ведении журнала для конкретного приложения, т.к. дает возможность отследить путь к источнику ошибки.

Свойство `HelpLink`

Хотя свойства `TargetSite` и `StackTrace` позволяют программистам выяснить, почему возникло конкретное исключение, информация подобного рода не особенно полезна для пользователей. Как уже было показано, с помощью свойства `System.Exception.Message` можно извлечь читабельную информацию и отобразить ее конечному пользователю. Вдобавок можно установить свойство `HelpLink` для указания на специальный URL или стандартный справочный файл, где приводятся более подробные сведения о проблеме.

По умолчанию значением свойства `HelpLink` является пустая строка. Обновите исключение с использованием инициализации объектов, чтобы предоставить более интересное значение. Ниже показан модифицированный код метода `Car.Accelerate()`:

```
public void Accelerate(int delta)
{
  if (_carIsDead)
  {
    Console.WriteLine("{0} is out of order...", PetName);
  }
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      CurrentSpeed = 0;
      _carIsDead = true;
      // Использовать ключевое слово throw для генерации
      // исключения и возврата в вызывающий код.
      throw new Exception($"{PetName} has overheated!")
      {
        HelpLink = "http://www.CarsRUs.com"
      };
    }
    Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

Теперь можно обновить логику в блоке `catch` для вывода на консоль информации из свойства `HelpLink`:

```
catch(Exception e)
{
  ...
  Console.WriteLine("Help Link: {0}", e.HelpLink);
}
```

Свойство `Data`

Свойство `Data` класса `System.Exception` позволяет заполнить объект исключения подходящей вспомогательной информацией (такой как отметка времени). Свойство `Data` возвращает объект, который реализует интерфейс по имени `IDictionary`, определенный в пространстве имен `System.Collections`. В главе 8 исследуется роль программирования на основе интерфейсов, а также рассматривается пространство имен `System.Collections`. В текущий момент важно понимать лишь то, что словарные коллекции позволяют создавать наборы значений, извлекаемых по ключу. Взгляните на очередное изменение метода `Car.Accelerate()`:

```
public void Accelerate(int delta)
{
  if (_carIsDead)
  {
    Console.WriteLine("{0} is out of order...", PetName);
  }
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      Console.WriteLine("{0} has overheated!", PetName);
      CurrentSpeed = 0;
      _carIsDead = true;
      // Использовать ключевое слово throw для генерации
      // исключения и возврата в вызывающий код.
      throw new Exception($"{PetName} has overheated!")
      {
        HelpLink = "http://www.CarsRUs.com",
        Data = {
          {"TimeStamp",$"The car exploded at {DateTime.Now}"},
          {"Cause","You have a lead foot."}
        }
      };
    }
    Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

С целью успешного прохода по парам "ключ-значение" добавьте директиву `using` для пространства имен `System.Collections`, т.к. в файле с операторами верхнего уровня будет применяться тип `DictionaryEntry`:

```
using System.Collections;
```

Затем обновите логику в блоке `catch`, чтобы обеспечить проверку значения, возвращаемого из свойства `Data`, на равенство `null` (т.е. стандартному значению). После этого свойства `Key` и `Value` типа `DictionaryEntry` используются для вывода специальных данных на консоль:

```
catch (Exception e)
{
  ...
  Console.WriteLine("\n-> Custom Data:");
  foreach (DictionaryEntry de in e.Data)
  {
    Console.WriteLine("-> {0}: {1}", de.Key, de.Value);
  }
}
```

Вот как теперь выглядит финальный вывод программы:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100
*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
Stack: at SimpleException.Car.Accelerate(Int32 delta) ...
       at SimpleException.Program.Main(String[] args) ...
Help Link: http://www.CarsRUs.com

-> Custom Data:
-> TimeStamp: The car exploded at 3/15/2020 16:22:59
-> Cause: You have a lead foot.

***** Out of exception logic *****
```

Свойство `Data` удобно в том смысле, что оно позволяет упаковывать специальную информацию об ошибке, не требуя построения нового типа класса для расширения базового класса `Exception`. Тем не менее, каким бы полезным ни было свойство `Data`, разработчики все равно обычно строят строго типизированные классы исключений, которые поддерживают специальные данные, применяя строго типизированные свойства.

Такой подход позволяет вызывающему коду перехватывать конкретный тип, производный от `Exception`, а не углубляться в коллекцию данных с целью получения дополнительных деталей. Чтобы понять, как это работает, необходимо разобраться с разницей между исключениями уровня системы и уровня приложения.

Исключения уровня системы (System.SystemException)

В библиотеках базовых классов .NET 5 определено много классов, которые в конечном итоге являются производными от System.Exception.

Например, в пространстве имен System определены основные объекты исключений, такие как ArgumentOutOfRangeException, IndexOutOfRangeException, StackOverflowException и т.п. В других пространствах имен есть исключения, которые отражают поведение этих пространств имен. Например, в System.Drawing.Printing определены исключения, связанные с печатью, в System.IO — исключения, возникающие во время ввода-вывода, в System.Data — исключения, специфичные для баз данных, и т.д.

Исключения, которые генерируются самой платформой .NET 5, называются *системными исключениями*. Такие исключения в общем случае рассматриваются как неисправимые фатальные ошибки. Системные исключения унаследованы прямо от базового класса System.SystemException, который в свою очередь порожден от System.Exception (а тот — от класса System.Object):

```
public class SystemException : Exception
{
  // Разнообразные конструкторы.
}
```

Учитывая, что тип System.SystemException не добавляет никакой дополнительной функциональности кроме набора специальных конструкторов, вас может интересовать, по какой причине он вообще существует. Попросту говоря, когда тип исключения является производным от System.SystemException, то есть возможность выяснить, что исключение сгенерировала исполняющая среда .NET 5, а не кодовая база выполняющегося приложения. Это довольно легко проверить, используя ключевое слово is:

```
// Верно! NullReferenceException является SystemException.
NullReferenceException nullRefEx = new NullReferenceException();
Console.WriteLine(
  "NullReferenceException is-a SystemException? : {0}",
  nullRefEx is SystemException);
```

Исключения уровня приложения (System.ApplicationException)

Поскольку все исключения .NET 5 являются типами классов, вы можете создавать собственные исключения, специфичные для приложения. Однако из-за того, что базовый класс System.SystemException представляет исключения, генерируемые исполняющей средой, может сложиться впечатление, что вы должны порождать свои специальные исключения от типа System.Exception. Конечно, можно поступать и так, но взамен их лучше наследовать от класса System.ApplicationException:

```
public class ApplicationException : Exception
{
  // Разнообразные конструкторы.
}
```

Как и в SystemException, кроме набора конструкторов никаких дополнительных членов в классе ApplicationException не определено. С точки зрения функциональности единственная цель класса System.ApplicationException состоит в идентификации источника ошибки. При обработке исключения, производного от System.ApplicationException, можно предполагать, что исключение было сгенерировано кодовой базой выполняющегося приложения, а не библиотеками базовых классов .NET Core либо исполняющей средой .NET 5.

Построение специальных исключений, способ первый

Наряду с тем, что для сигнализации об ошибке во время выполнения можно всегда генерировать экземпляры System.Exception (как было показано в первом примере), иногда предпочтительнее создавать *строго типизированное исключение*, которое представляет уникальные детали, связанные с текущей проблемой.

Например, предположим, что вы хотите построить специальное исключение (по имени CarIsDeadException) для представления ошибки, которая возникает из-за увеличения скорости обреченного на выход из строя автомобиля. Первым делом создается новый класс, унаследованный от System.Exception/System.ApplicationException (по соглашению имена всех классов исключений заканчиваются суффиксом Exception).

На заметку! Согласно правилу все специальные классы исключений должны быть определены как открытые (вспомните, что стандартным модификатором доступа для невложенных типов является internal). Причина в том, что исключения часто передаются за границы сборок и потому должны быть доступны вызывающей кодовой базе.

Создайте новый проект консольного приложения по имени CustomException, скопируйте в него предыдущие файлы Car.cs и Radio.cs и измените название пространства имен, в котором определены типы Car и Radio, с SimpleException на CustomException.

Затем добавьте в проект новый файл по имени CarIsDeadException.cs и поместите в него следующее определение класса:

```
using System;
namespace CustomException
{
  // Это специальное исключение описывает детали условия
  // выхода автомобиля из строя.
  // (Не забывайте, что можно также просто расширить класс Exception.)
  public class CarIsDeadException : ApplicationException
  {
  }
}
```

Как и с любым классом, вы можете создавать произвольное количество специальных членов, к которым можно обращаться внутри блока catch в вызывающем коде. Кроме того, вы можете также переопределять любые виртуальные члены, определенные в родительских классах. Например, вы могли бы реализовать CarIsDeadException, переопределив виртуальное свойство Message.

Вместо заполнения словаря данных (через свойство Data) при генерировании исключения конструктор позволяет указывать отметку времени и причину возникновения ошибки. Наконец, отметку времени и причину возникновения ошибки можно получить с применением строго типизированных свойств:

```
public class CarIsDeadException : ApplicationException
{
  private string _messageDetails = String.Empty;
  public DateTime ErrorTimeStamp {get; set;}
  public string CauseOfError {get; set;}

  public CarIsDeadException(){}
  public CarIsDeadException(string message,
    string cause, DateTime time)
  {
    _messageDetails = message;
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }

  // Переопределить свойство Exception.Message.
  public override string Message
    => $"Car Error Message: {_messageDetails}";
}
```

Здесь класс `CarIsDeadException` поддерживает закрытое поле (`_messageDetails`), которое представляет данные, касающиеся текущего исключения; его можно устанавливать с использованием специального конструктора. Сгенерировать такое исключение в методе `Accelerate()` несложно. Понадобится просто создать, сконфигурировать и сгенерировать объект `CarIsDeadException`, а не `System.Exception`:

```
// Сгенерировать специальное исключение CarIsDeadException.
public void Accelerate(int delta)
{
  ...
  throw new CarIsDeadException(
    $"{PetName} has overheated!",
      "You have a lead foot", DateTime.Now)
  {
    HelpLink = "http://www.CarsRUs.com",
  };
  ...
}
```

Для перехвата такого входного исключения блок `catch` теперь можно модифицировать, чтобы в нем перехватывался конкретный тип `CarIsDeadException` (тем не менее, с учетом того, что `System.CarIsDeadException` "является" `System.Exception`, по-прежнему допустимо перехватывать `System.Exception`):

```
using System;
using CustomException;

Console.WriteLine("***** Fun with Custom Exceptions *****\n");
Car myCar = new Car("Rusty", 90);
try
{
  // Отслеживать исключение.
  myCar.Accelerate(50);
}
```

```
catch (CarIsDeadException e)
{
  Console.WriteLine(e.Message);
  Console.WriteLine(e.ErrorTimeStamp);
  Console.WriteLine(e.CauseOfError);
}
Console.ReadLine();
```

Итак, теперь, когда вы понимаете базовый процесс построения специального исключения, пришло время опереться на эти знания.

Построение специальных исключений, способ второй

В текущем классе `CarIsDeadException` переопределено виртуальное свойство `System.Exception.Message` с целью конфигурирования специального сообщения об ошибке и предоставлены два специальных свойства для учета дополнительных порций данных. Однако в реальности переопределять виртуальное свойство `Message` не обязательно, т.к. входное сообщение можно просто передать конструктору родительского класса:

```
public class CarIsDeadException : ApplicationException
{
  public DateTime ErrorTimeStamp { get; set; }
  public string CauseOfError { get; set; }

  public CarIsDeadException() { }

  // Передача сообщения конструктору родительского класса.
  public CarIsDeadException(string message, string cause, DateTime time)
    :base(message)
  {
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }
}
```

Обратите внимание, что на этот раз *не* объявляется строковая переменная для представления сообщения и *не* переопределяется свойство `Message`. Взамен нужный параметр просто передается конструктору базового класса. При таком проектном решении специальный класс исключения является всего лишь уникально именованным классом, производным от `System.ApplicationException` (с дополнительными свойствами в случае необходимости), который не переопределяет какие-либо члены базового класса.

Не удивляйтесь, если большинство специальных классов исключений (а то и все) будет соответствовать такому простому шаблону. Во многих случаях роль специального исключения не обязательно связана с предоставлением дополнительной функциональности помимо той, что унаследована от базовых классов. На самом деле цель в том, чтобы предложить *строго именованный тип*, который четко идентифицирует природу ошибки, благодаря чему клиент может реализовать отличающуюся логику обработки для разных типов исключений.

Построение специальных исключений, способ третий

Если вы хотите создать по-настоящему интересный специальный класс исключения, тогда необходимо обеспечить наличие у класса следующих характеристик:

- он является производным от класса Exception/ApplicationException;
- в нем определен стандартный конструктор;
- в нем определен конструктор, который устанавливает значение унаследованного свойства Message;
- в нем определен конструктор для обработки "внутренних исключений".

Чтобы завершить исследование специальных исключений, ниже приведена последняя версия класса CarIsDeadException, в которой реализованы все упомянутые выше специальные конструкторы (свойства будут такими же, как в предыдущем примере):

```
public class CarIsDeadException : ApplicationException
{
  private string _messageDetails = String.Empty;
  public DateTime ErrorTimeStamp {get; set;}
  public string CauseOfError {get; set;}

  public CarIsDeadException(){}
  public CarIsDeadException(string cause, DateTime time)
          : this(cause,time,string.Empty)
  {
  }
  public CarIsDeadException(string cause, DateTime time, string message) :
    this(cause,time,message, null)
  {
  }
  public CarIsDeadException(string cause, DateTime time,
                            string message, System.Exception inner)
    : base(message, inner)
  {
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }
}
```

Затем необходимо модифицировать метод Accelerate() с учетом обновленного специального исключения:

```
throw new CarIsDeadException("You have a lead foot",
  DateTime.Now,$"{PetName} has overheated!")
{
  HelpLink = "http://www.CarsRUs.com",
};
```

Поскольку создаваемые специальные исключения, следующие установившейся практике в .NET Core, на самом деле отличаются только своими именами, полезно знать, что среды Visual Studio и Visual Studio Code предлагает фрагмент кода, который автоматически генерирует новый класс исключения, отвечающий рекомендациям .NET. Для его активизации наберите exc и нажмите клавишу <Tab> (в Visual Studio нажмите <Tab> два раза).

Обработка множества исключений

В своей простейшей форме блок try сопровождается единственным блоком catch. Однако в реальности часто приходится сталкиваться с ситуациями, когда операторы внутри блока try могут генерировать *многочисленные* исключения. Создайте новый проект консольного приложения на C# по имени ProcessMultipleExpceptions, скопируйте в него файлы Car.cs, Radio.cs и CarIsDeadException.cs из предыдущего проекта CustomException и надлежащим образом измените название пространства имен.

Затем модифицируйте метод Accelerate() класса Car так, чтобы он генерировал еще и предопределенное в библиотеках базовых классов исключение ArgumentOutOfRangeException, если передается недопустимый параметр (которым будет считаться любое значение меньше нуля). Обратите внимание, что конструктор этого класса исключения принимает имя проблемного аргумента в первом параметре типа string, за которым следует сообщение с описанием ошибки.

```
// Перед продолжением проверить аргумент на предмет допустимости.
public void Accelerate(int delta)
{
  if (delta < 0)
  {
    throw new ArgumentOutOfRangeException(nameof(delta),
      "Speed must be greater than zero");
    // Значение скорости должно быть больше нуля!
  }
  ...
}
```

На заметку! Операция nameof() возвращает строку, представляющую имя объекта, т.е. переменную delta в рассматриваемом примере. Такой прием позволяет безопасно ссылаться на объекты, методы и переменные C#, когда требуются их строковые версии.

Теперь логика в блоке catch может реагировать на каждый тип исключения специфическим образом:

```
using System;
using System.IO;
using ProcessMultipleExceptions;

Console.WriteLine("***** Handling Multiple Exceptions *****\n");
Car myCar = new Car("Rusty", 90);
try
{
  // Вызвать исключение выхода за пределы диапазона аргумента.
  myCar.Accelerate(-10);
}
catch (CarIsDeadException e)
{
  Console.WriteLine(e.Message);
}
catch (ArgumentOutOfRangeException e)
{
  Console.WriteLine(e.Message);
}
Console.ReadLine();
```

При написании множества блоков `catch` вы должны иметь в виду, что когда исключение сгенерировано, оно будет обрабатываться первым подходящим блоком `catch`. Чтобы проиллюстрировать, что означает "первый подходящий" блок `catch`, модифицируйте предыдущий код, добавив еще один блок `catch`, который пытается обработать все остальные исключения кроме `CarIsDeadException` и `ArgumentOutOfRangeException` путем перехвата общего типа `System.Exception`:

```
// Этот код не скомпилируется!
Console.WriteLine("***** Handling Multiple Exceptions *****\n");
Car myCar = new Car("Rusty", 90);
try
{
  // Вызвать исключение выхода за пределы диапазона аргумента.
  myCar.Accelerate(-10);
}
catch(Exception e)
{
  // Обработать все остальные исключения?
  Console.WriteLine(e.Message);
}
catch (CarIsDeadException e)
{
  Console.WriteLine(e.Message);
}
catch (ArgumentOutOfRangeException e)
{
  Console.WriteLine(e.Message);
}
Console.ReadLine();
```

Представленная выше логика обработки исключений приводит к возникновению ошибок на этапе компиляции. Проблема в том, что первый блок `catch` способен обрабатывать *любые* исключения, производные от `System.Exception` (с учетом отношения "является"), в том числе `CarIsDeadException` и `ArgumentOutOfRangeException`. Следовательно, два последних блока `catch` в принципе недостижимы!

Запомните эмпирическое правило: блоки `catch` должны быть структурированы так, чтобы первый `catch` перехватывал наиболее специфическое исключение (т.е. производный тип, расположенный ниже всех в цепочке наследования типов исключений), а последний `catch` — самое общее исключение (т.е. базовый класс имеющейся цепочки наследования; `System.Exception` в данном случае).

Таким образом, если вы хотите определить блок `catch`, который будет обрабатывать любые исключения помимо `CarIsDeadException` и `ArgumentOutOfRangeException`, то можно было бы написать следующий код:

```
// Этот код скомпилируется без проблем.
Console.WriteLine("***** Handling Multiple Exceptions *****\n");
Car myCar = new Car("Rusty", 90);
try
{
  // Вызвать исключение выхода за пределы диапазона аргумента.
  myCar.Accelerate(-10);
}
```

```
catch (CarIsDeadException e)
{
  Console.WriteLine(e.Message);
}
catch (ArgumentOutOfRangeException e)
{
  Console.WriteLine(e.Message);
}
// Этот блок будет перехватывать все остальные исключения
// помимо CarIsDeadException и ArgumentOutOfRangeException.
catch (Exception e)
{
  Console.WriteLine(e.Message);
}
Console.ReadLine();
```

На заметку! Везде, где только возможно, отдавайте предпочтение перехвату специфичных классов исключений, а не общего класса `System.Exception`. Хотя может показаться, что это упрощает жизнь в краткосрочной перспективе (поскольку охватывает все исключения, которые пока не беспокоят), в долгосрочной перспективе могут возникать странные аварийные отказы во время выполнения, т.к. в коде не была предусмотрена непосредственная обработка более серьезной ошибки. Не забывайте, что финальный блок `catch`, который работает с `System.Exception`, на самом деле имеет тенденцию быть чрезвычайно общим.

Общие операторы `catch`

В языке C# также поддерживается "общий" контекст `catch`, который не получает явно объект исключения, сгенерированный заданным членом:

```
// Общий оператор catch.
Console.WriteLine("***** Handling Multiple Exceptions *****\n");
Car myCar = new Car("Rusty", 90);
try
{
  myCar.Accelerate(90);
}
catch
{
  Console.WriteLine("Something bad happened...");
                  // Произошло что-то плохое...
}
Console.ReadLine();
```

Очевидно, что это не самый информативный способ обработки исключений, поскольку нет никакой возможности для получения содержательных данных о возникшей ошибке (таких как имя метода, стек вызовов или специальное сообщение). Тем не менее, в C# такая конструкция разрешена, потому что она может быть полезной, когда требуется обрабатывать все ошибки в обобщенной манере.

Повторная генерация исключений

Внутри логики блока `try` перехваченное исключение разрешено *повторно сгенерировать* для передачи вверх по стеку вызовов предшествующему вызывающему коду. Для этого просто используется ключевое слово `throw` в блоке `catch`. В итоге исключение передается вверх по цепочке вызовов, что может оказаться полезным, если блок `catch` способен обработать текущую ошибку только частично:

```
// Передача ответственности.
...
try
{
  // Логика увеличения скорости автомобиля...
}
catch(CarIsDeadException e)
{
  // Выполнить частичную обработку этой ошибки и передать ответственность.
  throw;
}
...
```

Имейте в виду, что в данном примере кода конечным получателем исключения `CarIsDeadException` будет исполняющая среда .NET 5, т.к. операторы верхнего уровня генерируют его повторно. По указанной причине конечному пользователю будет отображаться системное диалоговое окно с информацией об ошибке. Обычно вы будете повторно генерировать частично обработанное исключение для передачи вызывающему коду, который имеет возможность обработать входное исключение более элегантным образом.

Также обратите внимание на неявную повторную генерацию объекта `CarIsDeadException` с помощью ключевого слова `throw` без аргументов. Дело в том, что здесь не создается новый объект исключения, а просто передается исходный объект исключения (со всей исходной информацией). Это позволяет сохранить контекст первоначального целевого объекта.

Внутренние исключения

Как нетрудно догадаться, вполне возможно, что исключение сгенерируется во время обработки другого исключения. Например, пусть вы обрабатываете исключение `CarIsDeadException` внутри отдельного блока `catch` и в ходе этого процесса пытаетесь записать данные трассировки стека в файл `carErrors.txt` на диске C: (для получения доступа к типам, связанным с вводом-выводом, потребуется добавить директиву `using` с пространством имен `System.IO`):

```
catch(CarIsDeadException e)
{
  // Попытка открытия файла carErrors.txt, расположенного на диске C:.
  FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
  ...
}
```

Если указанный файл на диске C: отсутствует, тогда вызов метода `File.Open()` приведет к генерации исключения `FileNotFoundException`! Позже в книге, когда мы будем подробно рассматривать пространство имен `System.IO`, вы узнаете, как

программно определить, существует ли файл на жестком диске, перед попыткой его открытия (тем самым вообще избегая исключения). Однако чтобы не отклоняться от темы исключений, мы предположим, что такое исключение было сгенерировано.

Когда во время обработки исключения вы сталкиваетесь с еще одним исключением, установившаяся практика предусматривает обязательное сохранение нового объекта исключения как "внутреннего исключения" в новом объекте того же типа, что и исходное исключение. Причина, по которой необходимо создавать новый объект обрабатываемого исключения, связана с тем, что единственным способом документирования внутреннего исключения является применение параметра конструктора. Взгляните на следующий код:

```
using System.IO;
// Обновление обработчика исключений.
catch (CarIsDeadException e)
{
  try
  {
    FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
    ...
  }
  catch (Exception e2)
  {
    // Следующая строка приведет к ошибке на этапе компиляции,
    // т.к. InnerException допускает только чтение.
    // e.InnerException = e2;
    // Сгенерировать исключение, которое записывает новое
    // исключение, а также сообщение из первого исключения.
    throw new CarIsDeadException(
      e.CauseOfError, e.ErrorTimeStamp, e.Message, e2);
 }
}
```

Обратите внимание, что в данном случае конструктору `CarIsDeadException` во втором параметре передается объект `FileNotFoundException`. После настройки этого нового объекта он передается вверх по стеку вызовов следующему вызывающему коду, которым в рассматриваемой ситуации будут операторы верхнего уровня.

Поскольку после операторов верхнего уровня нет "следующего вызывающего кода", который мог бы перехватить исключение, пользователю будет отображено системное диалоговое окно с сообщением об ошибке. Подобно повторной генерации исключения запись внутренних исключений обычно полезна, только если вызывающий код способен обработать исключение более элегантно. В таком случае внутри логики `catch` вызывающего кода можно использовать свойство `InnerException` для извлечения деталей внутреннего исключения.

Блок `finally`

В области действия `try/catch` можно также определять дополнительный блок `finally`. Целью блока `finally` является обеспечение того, что заданный набор операторов будет выполняться *всегда* независимо от того, возникло исключение (любого типа) или нет. Для иллюстрации предположим, что перед завершением программы радиоприемник в автомобиле должен всегда выключаться вне зависимости от обрабатываемого исключения:

```
Console.WriteLine("***** Handling Multiple Exceptions *****\n");
Car myCar = new Car("Rusty", 90);
myCar.CrankTunes(true);
try
{
  // Логика, связанная с увеличением скорости автомобиля.
}
catch(CarIsDeadException e)
{
  // Обработать объект CarIsDeadException.
}
catch(ArgumentOutOfRangeException e)
{
  // Обработать объект ArgumentOutOfRangeException.
}
catch(Exception e)
{
  // Обработать любой другой объект Exception.
}
finally
{
  // Это код будет выполняться всегда независимо
  // от того, возникало исключение или нет.
  myCar.CrankTunes(false);
}
Console.ReadLine();
```

Если вы не определите блок `finally`, то в случае генерации исключения радиоприемник не выключится (что может быть или не быть проблемой). В более реалистичном сценарии, когда необходимо освободить объекты, закрыть файл либо отсоединиться от базы данных (или чего-то подобного), блок `finally` представляет собой подходящее место для выполнения надлежащей очистки.

Фильтры исключений

В версии C# 6 появилась новая конструкция, которая может быть помещена в блок `catch` посредством ключевого слова `when`. В случае ее добавления появляется возможность обеспечить выполнение операторов внутри блока `catch` только при удовлетворении некоторого условия в коде. Выражение условия должно давать в результате булевское значение (`true` или `false`) и может быть указано с применением простого выражения в самом определении `when` либо за счет вызова дополнительного метода в коде. Коротко говоря, такой подход позволяет добавлять "фильтры" к логике исключения.

Взгляните на показанную ниже модифицированную логику исключения. Здесь к обработчику `CarIsDeadException` добавлена конструкция `when`, которая гарантирует, что данный блок `catch` никогда не будет выполняться по пятницам (конечно, пример надуман, но кто захочет разбирать автомобиль на выходные?). Обратите внимание, что одиночное булевское выражение в конструкции `when` должно быть помещено в круглые скобки.

```
catch (CarIsDeadException e)
  when (e.ErrorTimeStamp.DayOfWeek != DayOfWeek.Friday)
{
```

```
    // Выводится, только если выражение в конструкции when
    // вычисляется как true.
    Console.WriteLine("Catching car is dead!");
    Console.WriteLine(e.Message);
}
```

Рассмотренный пример был надуманным, а более реалистичное использование фильтра исключений предусматривает перехват экземпляров `SystemException`. Скажем, пусть ваш код сохраняет информацию в базу данных и генерируется общее исключение. Изучив сообщение и детали исключения, вы можете создать специфические обработчики, основанные на том, что конкретно было причиной исключения.

Отладка необработанных исключений с использованием Visual Studio

Среда Visual Studio предлагает набор инструментов, которые помогают отлаживать необработанные исключения. Предположим, что вы увеличили скорость объекта `Car` до значения, превышающего максимум, но на этот раз не позаботились о помещении вызова внутрь блока `try`:

```
Car myCar = new Car("Rusty", 90);
myCar.Accelerate(100);
```

Если вы запустите сеанс отладки в Visual Studio (выбрав пункт меню Debug⇨Start (Отладка⇨Начать)), то во время генерации необработанного исключения произойдет автоматический останов. Более того, откроется окно (рис. 7.1), отображающее значение свойства `Message`.

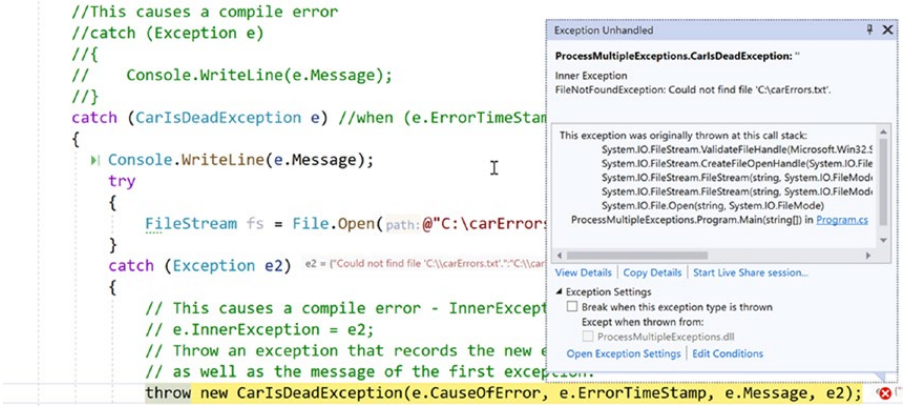

Рис. 7.1. Отладка необработанных специальных исключений в Visual Studio

На заметку! Если вы не обработали исключение, сгенерированное каким-то методом из библиотек базовых классов .NET 5, тогда отладчик Visual Studio остановит выполнение на операторе, который вызвал проблемный метод.

Щелкнув в этом окне на ссылке View Detail (Показать подробности), вы обнаружите подробную информацию о состоянии объекта (рис. 7.2).

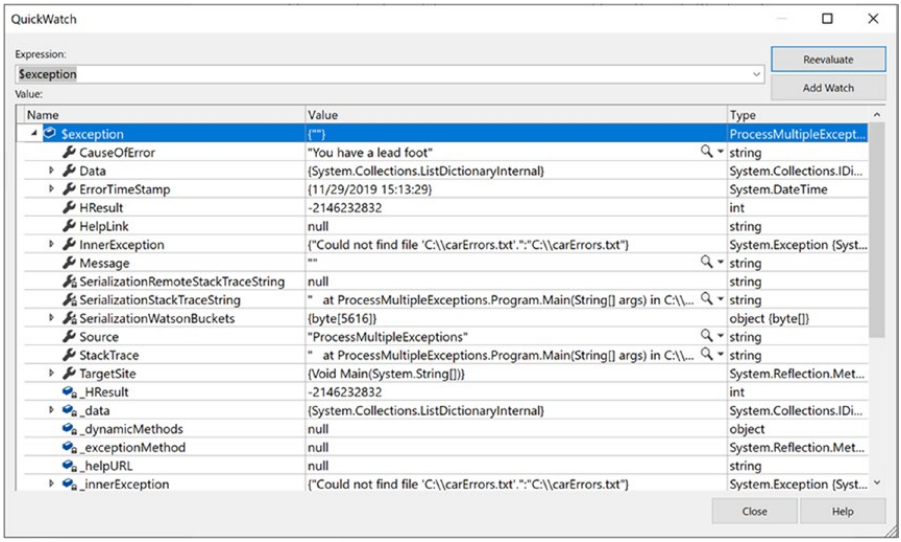

Рис. 7.2. Просмотр деталей исключения

Резюме

В главе была раскрыта роль структурированной обработки исключений. Когда методу необходимо отправить объект ошибки вызывающему коду, он должен создать, сконфигурировать и сгенерировать специфичный объект производного от `System.Exception` типа посредством ключевого слова `throw` языка C#. Вызывающий код может обрабатывать любые входные исключения с применением ключевого слова `catch` и необязательного блока `finally`. В версии C# 6 появилась возможность создавать фильтры исключений с использованием дополнительного ключевого слова `when`, а в версии C# 7 расширен перечень мест, где можно генерировать исключения.

Когда вы строите собственные специальные исключения, то в конечном итоге создаете класс, производный от класса `System.ApplicationException`, который обозначает исключение, генерируемое текущим выполняющимся приложением. В противоположность этому объекты ошибок, производные от класса `System.SystemException`, представляют критические (и фатальные) ошибки, генерируемые исполняющей средой .NET 5. Наконец, в главе были продемонстрированы разнообразные инструменты среды Visual Studio, которые можно применять для создания специальных исключений (согласно установившейся практике .NET), а также для отладки необработанных исключений.

ГЛАВА 8

Работа с интерфейсами

Материал настоящей главы опирается на ваши текущие знания объектно-ориентированной разработки и посвящен теме программирования на основе интерфейсов. Вы узнаете, как определять и реализовывать интерфейсы, а также ознакомитесь с преимуществами построения типов, которые поддерживают несколько линий поведения. В ходе изложения обсуждаются связанные темы, такие как получение ссылок на интерфейсы, явная реализация интерфейсов и построение иерархий интерфейсов. Будет исследовано несколько стандартных интерфейсов, определенных внутри библиотек базовых классов .NET Core. Кроме того, раскрываются новые средства C# 8, связанные с интерфейсами, в том числе стандартные методы интерфейсов, статические члены и модификаторы доступа. Вы увидите, что специальные классы и структуры могут реализовывать эти предопределенные интерфейсы для поддержки ряда полезных аспектов поведения, включая клонирование, перечисление и сортировку объектов.

Понятие интерфейсных типов

Первым делом давайте ознакомимся с формальным определением *интерфейсного типа*, которое с появлением версии C# 8 изменилось. До выхода C# 8 интерфейс был не более чем именованным набором *абстрактных членов*. Вспомните из главы 6, что абстрактные методы являются чистым протоколом, поскольку они не предоставляют свои стандартные реализации. Специфичные члены, определяемые интерфейсом, зависят от того, какое точно поведение он моделирует. Другими словами, интерфейс выражает *поведение*, которое заданный класс или структура может избрать для поддержки. Более того, далее в главе вы увидите, что класс или структура может реализовывать столько интерфейсов, сколько необходимо, и посредством этого поддерживать по существу множество линий поведения.

Средство стандартных методов интерфейсов, введенное в C# 8.0, позволяет методам интерфейса содержать реализацию, которая может переопределяться или не переопределяться в классе реализации. Более подробно о таком средстве речь пойдет позже в главе.

Как вы наверняка догадались, библиотеки базовых классов .NET Core поставляются с многочисленными предопределенными интерфейсными типами, которые реализуются разнообразными классами и структурами. Например, в главе 21 будет показано, что инфраструктура ADO.NET содержит множество поставщиков данных, которые позволяют взаимодействовать с определенной системой управления базами данных. Таким образом, в ADO.NET на выбор доступен обширный набор классов подключений (`SqlConnection`, `OleDbConnection`, `OdbcConnection` и т.д.). Вдобавок независимые

поставщики баз данных (а также многие проекты с открытым кодом) предлагают библиотеки .NET Core для взаимодействия с большим числом других баз данных (MySQL, Oracle и т.д.), которые содержат объекты, реализующие упомянутые интерфейсы.

Невзирая на тот факт, что каждый класс подключения имеет уникальное имя, определен в отдельном пространстве имен и (в некоторых случаях) упакован в отдельную сборку, все они реализуют общий интерфейс под названием `IDbConnection`:

```
// Интерфейс IDbConnection определяет общий набор членов,
// поддерживаемый всеми классами подключения.
public interface IDbConnection : IDisposable
{
  // Методы.
  IDbTransaction BeginTransaction();
  IDbTransaction BeginTransaction(IsolationLevel il);
  void ChangeDatabase(string databaseName);
  void Close();
  IDbCommand CreateCommand();
  void Open();

  // Свойства.
  string ConnectionString { get; set;}
  int ConnectionTimeout { get; }
  string Database { get; }
  ConnectionState State { get; }
}
```

На заметку! По соглашению имена интерфейсов .NET снабжаются префиксом в виде заглавной буквы `I`. При создании собственных интерфейсов рекомендуется также следовать этому соглашению.

В настоящий момент детали того, что делают члены интерфейса `IDbConnection`, не важны. Просто запомните, что в `IDbConnection` определен набор членов, которые являются общими для всех классов подключений ADO.NET. В итоге каждый класс подключения гарантированно поддерживает такие члены, как `Open()`, `Close()`, `CreateCommand()` и т.д. Кроме того, поскольку методы интерфейса `IDbConnection` всегда абстрактные, в каждом классе подключения они могут быть реализованы уникальным образом.

В оставшихся главах книги вы встретите десятки интерфейсов, поставляемых в библиотеках базовых классов .NET Core. Вы увидите, что эти интерфейсы могут быть реализованы в собственных специальных классах и структурах для определения типов, которые тесно интегрированы с платформой. Вдобавок, как только вы оцените полезность интерфейсных типов, вы определенно найдете причины для построения собственных таких типов.

Сравнение интерфейсных типов и абстрактных базовых классов

Учитывая материалы главы 6, интерфейсный тип может выглядеть кое в чем похожим на абстрактный базовый класс. Вспомните, что когда класс помечен как абстрактный, он *может* определять любое количество абстрактных членов для предоставления полиморфного интерфейса всем производным типам. Однако даже если класс действительно определяет набор абстрактных членов, он также может определять любое количество конструкторов, полей данных, неабстрактных членов (с реали-

зацией) и т.д. Интерфейсы (до C# 8.0) содержат *только* определения членов. Начиная с версии C# 8, интерфейсы способны содержать определения членов (вроде абстрактных членов), члены со стандартными реализациями (наподобие виртуальных членов) и статические члены. Есть только два реальных отличия: интерфейсы не могут иметь нестатические конструкторы, а класс может реализовывать множество интерфейсов. Второй аспект обсуждается следующим.

Полиморфный интерфейс, устанавливаемый абстрактным родительским классом, обладает одним серьезным ограничением: члены, определенные абстрактным родительским классом, поддерживаются *только производными типами*. Тем не менее, в крупных программных системах часто разрабатываются многочисленные иерархии классов, не имеющие общего родителя кроме `System.Object`. Учитывая, что абстрактные члены в абстрактном базовом классе применимы только к производным типам, не существует какого-то способа конфигурирования типов в разных иерархиях для поддержки одного и того же полиморфного интерфейса. Для начала создайте новый проект консольного приложения по имени `CustomInterfaces`. Добавьте к проекту следующий абстрактный класс:

```
namespace CustomInterfaces
{
  public abstract class CloneableType
  {
    // Поддерживать этот "полиморфный интерфейс"
    // могут только производные типы.
    // Классы в других иерархиях не имеют доступа
    // к данному абстрактному члену.
    public abstract object Clone();
  }
}
```

При таком определении поддерживать метод `Clone()` способны только классы, расширяющие `CloneableType`. Если создается новый набор классов, которые не расширяют данный базовый класс, то извлечь пользу от такого полиморфного интерфейса не удастся. К тому же вы можете вспомнить, что язык C# не поддерживает множественное наследование для классов. По этой причине, если вы хотите создать класс `MiniVan`, который является и `Car`, и `CloneableType`, то поступить так, как показано ниже, не удастся:

```
// Недопустимо! Множественное наследование для классов в C# невозможно.
public class MiniVan : Car, CloneableType
{
}
```

Несложно догадаться, что на помощь здесь приходят интерфейсные типы. После того как интерфейс определен, он может быть реализован любым классом либо структурой, в любой иерархии и внутри любого пространства имен или сборки (написанной на любом языке программирования .NET Core). Как видите, интерфейсы являются *чрезвычайно* полиморфными. Рассмотрим стандартный интерфейс .NET Core под названием `ICloneable`, определенный в пространстве имен `System`. В нем определен единственный метод по имени `Clone()`:

```
public interface ICloneable
{
  object Clone();
}
```

Во время исследования библиотек базовых классов .NET Core вы обнаружите, что интерфейс ICloneable реализован очень многими на вид несвязанными типами (System.Array, System.Data.SqlClient.SqlConnection, System.OperatingSystem, System.String и т.д.). Хотя указанные типы не имеют общего родителя (кроме System.Object), их можно обрабатывать полиморфным образом посредством интерфейсного типа ICloneable.

Первым делом поместите в файл Program.cs следующий код:

```
using System;
using CustomInterfaces;

Console.WriteLine("***** A First Look at Interfaces *****\n");
CloneableExample();
```

Далее добавьте к операторам верхнего уровня показанную ниже локальную функцию по имени CloneMe(), которая принимает параметр типа ICloneable, что позволит передавать любой объект, реализующий указанный интерфейс:

```
static void CloneableExample()
{
  // Все эти классы поддерживают интерфейс ICloneable.
  string myStr = "Hello";
  OperatingSystem unixOS =
    new OperatingSystem(PlatformID.Unix, new Version());

  // Следовательно, все они могут быть переданы методу,
  // принимающему параметр типа ICloneable.
  CloneMe(myStr);
  CloneMe(unixOS);

  static void CloneMe(ICloneable c)
  {
    // Клонировать то, что получено, и вывести имя.
    object theClone = c.Clone();
    Console.WriteLine("Your clone is a: {0}",
    theClone.GetType().Name);
  }
}
```

После запуска приложения в окне консоли выводится имя каждого класса, полученное с помощью метода GetType(), который унаследован от System.Object. Как будет объясняться в главе 17, этот метод позволяет выяснить строение любого типа во время выполнения. Вот вывод предыдущей программы:

```
***** A First Look at Interfaces *****

Your clone is a: String
Your clone is a: OperatingSystem
```

Еще одно ограничение абстрактных базовых классов связано с тем, что *каждый производный тип* должен предоставлять реализацию для всего набора абстрактных членов. Чтобы увидеть, в чем заключается проблема, вспомним иерархию фигур, которая была определена в главе 6. Предположим, что в базовом классе Shape определен новый абстрактный метод по имени GetNumberOfPoints(), который позволяет производным типам возвращать количество вершин, требуемых для визуализации фигуры:

```
namespace CustomInterfaces
{
  abstract class Shape
  {
    ...
    // Теперь этот метод обязан поддерживать каждый производный класс!
    public abstract byte GetNumberOfPoints();
  }
}
```

Очевидно, что единственным классом, который в принципе имеет вершины, будет `Hexagon`. Однако теперь из-за внесенного обновления *каждый* производный класс (`Circle`, `Hexagon` и `ThreeDCircle`) обязан предоставить конкретную реализацию метода `GetNumberOfPoints()`, даже если в этом нет никакого смысла. И снова интерфейсный тип предлагает решение. Если вы определите интерфейс, который представляет поведение "наличия вершин", то можно будет просто подключить его к классу `Hexagon`, оставив классы `Circle` и `ThreeDCircle` незатронутыми.

На заметку! Изменения интерфейсов в версии C# 8 являются, по всей видимости, наиболее существенными изменениями существующего языка за весь обозримый период. Как было ранее описано, новые возможности интерфейсов значительно приближают их функциональность к функциональности абстрактных классов с добавочной способностью классов реализовывать множество интерфейсов. В этой области рекомендуется проявлять надлежащую осторожность и здравый смысл. Один лишь факт, что вы можете что-то делать, вовсе не означает, что вы обязаны поступать так.

Определение специальных интерфейсов

Теперь, когда вы лучше понимаете общую роль интерфейсных типов, давайте рассмотрим пример определения и реализации специальных интерфейсов. Скопируйте файлы `Shape.cs`, `Hexagon.cs`, `Circle.cs` и `ThreeDCircle.cs` из решения `Shapes`, созданного в главе 6. Переименуйте пространство имен, в котором определены типы, связанные с фигурами, в `CustomInterface` (просто чтобы избежать импортирования в новый проект определений пространства имен). Добавьте в проект новый файл по имени `IPointy.cs`.

На синтаксическом уровне интерфейс определяется с использованием ключевого слова `interface` языка C#. В отличие от классов для интерфейсов никогда не задается базовый класс (даже `System.Object`; тем не менее, как будет показано позже в главе, можно задавать базовые интерфейсы). До выхода C# 8.0 для членов интерфейса не указывались модификаторы доступа (т.к. все члены интерфейса были неявно открытыми и абстрактными). В версии C# 8.0 можно также определять члены `private`, `internal`, `protected` и даже `static`, о чем пойдет речь далее в главе. Ниже приведен пример определения специального интерфейса в C#:

```
namespace CustomInterfaces
{
  // Этот интерфейс определяет поведение "наличия вершин".
  public interface IPointy
  {
    // Неявно открытый и абстрактный.
    byte GetNumberOfPoints();
  }
}
```

В интерфейсах в C# 8 нельзя определять поля данных или нестатические конструкторы. Таким образом, следующая версия интерфейса IPointy приведет к разнообразным ошибкам на этапе компиляции:

```
// Внимание! В этом коде полно ошибок!
public interface IPointy
{
  // Ошибка! Интерфейсы не могут иметь поля данных!
  public int numbOfPoints;

  // Ошибка! Интерфейсы не могут иметь нестатические конструкторы!
  public IPointy() { numbOfPoints = 0;}
}
```

В начальной версии интерфейса IPointy определен единственный метод. В интерфейсных типах допускается также определять любое количество прототипов свойств. Например, интерфейс IPointy можно было бы обновить, как показано ниже, закомментировав свойство для чтения-записи и добавив свойство только для чтения. Свойство Points заменяет метод GetNumberOfPoints().

```
// Поведение "наличия вершин" в виде свойства только для чтения.
public interface IPointy
{
  // Неявно public и abstract.
  // byte GetNumberOfPoints();

  // Свойство, поддерживающее чтение и запись,
  // в интерфейсе может выглядеть так:
  // string PropName { get; set; }

  // Тогда как свойство только для записи - так:
  byte Points { get; }
}
```

На заметку! Интерфейсные типы также могут содержать определения событий (глава 12) и индексаторов (глава 11).

Сами по себе интерфейсные типы совершенно бесполезны, поскольку выделять память для них, как делалось бы для класса или структуры, невозможно:

```
// Внимание! Выделять память для интерфейсных типов не допускается!
IPointy p = new IPointy();  // Ошибка на этапе компиляции!
```

Интерфейсы не привносят ничего особого до тех пор, пока не будут реализованы классом или структурой. Здесь IPointy представляет собой интерфейс, который выражает поведение "наличия вершин". Идея проста: одни классы в иерархии фигур (например, Hexagon) имеют вершины, в то время как другие (вроде Circle) — нет.

Реализация интерфейса

Когда функциональность класса (или структуры) решено расширить за счет поддержки интерфейсов, к определению добавляется список нужных интерфейсов, разделенных запятыми. Имейте в виду, что непосредственный базовый класс должен быть указан первым сразу после операции двоеточия. Если тип класса порождается напрямую от System.Object, тогда вы можете просто перечислить интерфейсы, поддерживаемые классом, т.к. компилятор C# будет считать, что типы расширяют System.

Object, если не задано иначе. К слову, поскольку структуры всегда являются производными от класса System.ValueType (см. главу 4), достаточно указать список интерфейсов после определения структуры. Взгляните на приведенные ниже примеры:

```
// Этот класс является производными от System.Object
// и реализует единственный интерфейс.
public class Pencil : IPointy
{...}
// Этот класс также является производными от System.Object
// и реализует единственный интерфейс.
public class SwitchBlade : object, IPointy
{...}
// Этот класс является производными от специального базового
// класса и реализует единственный интерфейс.
public class Fork : Utensil, IPointy
{...}
// Эта структура неявно является производной
// от System.ValueType и реализует два интерфейса.
public struct PitchFork : ICloneable, IPointy
{...}
```

Важно понимать, что для интерфейсных элементов, которые не содержат стандартной реализации, реализация интерфейса работает по плану "все или ничего". Поддерживающий тип не имеет возможности выборочно решать, какие члены он будет реализовывать. Учитывая, что интерфейс IPointy определяет единственное свойство только для чтения, накладные расходы невелики. Тем не менее, если вы реализуете интерфейс, который определяет десять членов (вроде показанного ранее IDbConnection), тогда тип отвечает за предоставление деталей для всех десяти абстрактных членов.

В текущем примере добавьте к проекту новый тип класса по имени Triangle, который "является" Shape и поддерживает IPointy. Обратите внимание, что реализация доступного только для чтения свойства Points (реализованного с использованием синтаксиса членов, сжатых до выражений) просто возвращает корректное количество вершин (т.е. 3):

```
using System;
namespace CustomInterfaces
{
  // Новый класс по имени Triangle, производный от Shape.
  class Triangle : Shape, IPointy
  {
    public Triangle() { }
    public Triangle(string name) : base(name) { }
    public override void Draw()
    {
      Console.WriteLine("Drawing {0} the Triangle", PetName);
    }
    // Реализация IPointy.
    // public byte Points
    // {
    //   get { return 3; }
    // }
    public byte Points => 3;
  }
}
```

Модифицируйте существующий тип `Hexagon`, чтобы он также поддерживал интерфейс `IPointy`:

```
using System;
namespace CustomInterfaces
{
  // Hexagon теперь реализует IPointy.
  class Hexagon : Shape, IPointy
  {
    public Hexagon(){ }
    public Hexagon(string name) : base(name){ }
    public override void Draw()
    {
      Console.WriteLine("Drawing {0} the Hexagon", PetName);
    }
    // Реализация IPointy.
    public byte Points => 6;
  }
}
```

Подводя итоги тому, что сделано к настоящему моменту, на рис. 8.1 приведена диаграмма классов в Visual Studio, где все совместимые с `IPointy` классы представлены с помощью популярной системы обозначений в виде "леденца на палочке". Еще раз обратите внимание, что `Circle` и `ThreeDCircle` не реализуют `IPointy`, поскольку такое поведение в этих классах не имеет смысла.

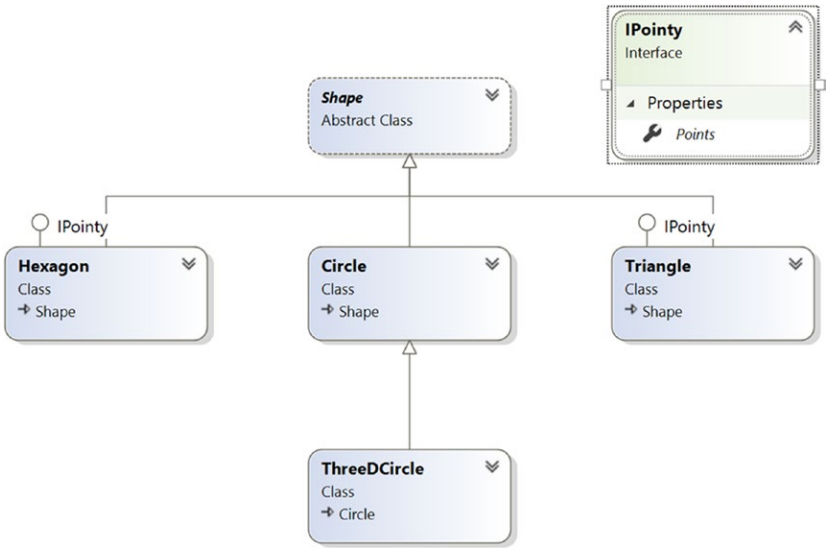

Рис. 8.1. Иерархия фигур, теперь с интерфейсами

На заметку! Чтобы скрыть или отобразить имена интерфейсов в визуальном конструкторе классов, щелкните правой кнопкой мыши на значке, представляющем интерфейс, и выберите в контекстном меню пункт Collapse (Свернуть) или Expand (Развернуть).

Обращение к членам интерфейса на уровне объектов

Теперь, имея несколько классов, которые поддерживают интерфейс `IPointy`, необходимо выяснить, каким образом взаимодействовать с новой функциональностью. Самый простой способ взаимодействия с функциональностью, предоставляемой заданным интерфейсом, заключается в обращении к его членам прямо на уровне объектов (при условии, что члены интерфейса не реализованы явно, о чем более подробно пойдет речь в разделе "Явная реализация интерфейсов" далее в главе). Например, взгляните на следующий код:

```
Console.WriteLine("***** Fun with Interfaces *****\n");
// Обратиться к свойству Points, определенному в интерфейсе IPointy.
Hexagon hex = new Hexagon();
Console.WriteLine("Points: {0}", hex.Points);
Console.ReadLine();
```

Данный подход нормально работает в этом конкретном случае, т.к. здесь точно известно, что тип `Hexagon` реализует упомянутый интерфейс и, следовательно, имеет свойство `Points`. Однако в других случаях определить, какие интерфейсы поддерживаются конкретным типом, может быть нереально. Предположим, что есть массив, содержащий 50 объектов совместимых с `Shape` типов, и только некоторые из них поддерживают интерфейс `IPointy`. Очевидно, что если вы попытаетесь обратиться к свойству `Points` для типа, который не реализует `IPointy`, то возникнет ошибка. Как же динамически определить, поддерживает ли класс или структура подходящий интерфейс?

Один из способов выяснить во время выполнения, поддерживает ли тип конкретный интерфейс, предусматривает применение явного приведения. Если тип не поддерживает запрашиваемый интерфейс, то генерируется исключение `InvalidCastException`. В случае подобного рода необходимо использовать структурированную обработку исключений:

```
...
// Перехватить возможное исключение InvalidCastException.
Circle c = new Circle("Lisa");
IPointy itfPt = null;
try
{
  itfPt = (IPointy)c;
  Console.WriteLine(itfPt.Points);
}
catch (InvalidCastException e)
{
  Console.WriteLine(e.Message);
}
Console.ReadLine();
```

Хотя можно было бы применить логику `try/catch` и надеяться на лучшее, в идеале хотелось бы определять, какие интерфейсы поддерживаются, до обращения к их членам. Давайте рассмотрим два способа, с помощью которых этого можно добиться.

Получение ссылок на интерфейсы: ключевое слово as

Для определения, поддерживает ли данный тип тот или иной интерфейс, можно использовать ключевое слово as, которое было представлено в главе 6. Если объект может трактоваться как указанный интерфейс, тогда возвращается ссылка на интересующий интерфейс, а если нет, то ссылка null. Таким образом, перед продолжением в коде необходимо реализовать проверку на предмет null:

```
...
// Можно ли hex2 трактовать как IPointy?
Hexagon hex2 = new Hexagon("Peter");
IPointy itfPt2 = hex2 as IPointy;
if(itfPt2 != null)
{
  Console.WriteLine("Points: {0}", itfPt2.Points);
}
else
{
  Console.WriteLine("OOPS! Not pointy...");  // Не реализует IPointy
}
Console.ReadLine();
```

Обратите внимание, что когда применяется ключевое слово as, отпадает необходимость в наличии логики try/catch, т.к. если ссылка не является null, то известно, что вызов происходит для действительной ссылки на интерфейс.

Получение ссылок на интерфейсы: ключевое слово is (обновление в версии 7.0)

Проверить, реализован ли нужный интерфейс, можно также с помощью ключевого слова is (о котором впервые упоминалось в главе 6). Если интересующий объект не совместим с указанным интерфейсом, тогда возвращается значение false. В случае предоставления в операторе имени переменной ей назначается надлежащий тип, что устраняет необходимость в проверке типа и выполнении приведения. Ниже показан обновленный предыдущий пример:

```
Console.WriteLine("***** Fun with Interfaces *****\n");
...
if(hex2 is IPointy itfPt3)
{
  Console.WriteLine("Points: {0}", itfPt3.Points);
}
else
{
  Console.WriteLine("OOPS! Not pointy...");
}
Console.ReadLine();
```

Стандартные реализации (нововведение в версии 8.0)

Как упоминалось ранее, в версии C# 8.0 методы и свойства интерфейса могут иметь стандартные реализации. Добавьте к проекту новый интерфейс по имени

IRegularPointy, предназначенный для представления многоугольника заданной формы. Вот код интерфейса:

```
namespace CustomInterfaces
{
  interface IRegularPointy : IPointy
  {
    int SideLength { get; set; }
    int NumberOfSides { get; set; }
    int Perimeter => SideLength * NumberOfSides;
  }
}
```

Добавьте к проекту новый файл класса по имени Square.cs, унаследуйте класс от базового класса Shape и реализуйте интерфейс IRegularPointy:

```
namespace CustomInterfaces
{
  class Square: Shape,IRegularPointy
  {
    public Square() { }
    public Square(string name) : base(name) { }
    // Метод Draw() поступает из базового класса Shape.
    public override void Draw()
    {
      Console.WriteLine("Drawing a square");
    }
    // Это свойство поступает из интерфейса IPointy.
    public byte Points => 4;
    // Это свойство поступает из интерфейса IRegularPointy.
    public int SideLength { get; set; }
    public int NumberOfSides { get; set; }
    // Обратите внимание, что свойство Perimeter не реализовано.
  }
}
```

Здесь мы невольно попали в первую "ловушку", связанную с использованием стандартных реализаций интерфейсов. Свойство Perimeter, определенное в интерфейсе IRegularPointy, в классе Square не определено, что делает его недоступным экземпляру класса Square. Чтобы удостовериться в этом, создайте новый экземпляр класса Square и выведите на консоль соответствующие значения:

```
Console.WriteLine("\n***** Fun with Interfaces *****\n");
...
var sq = new Square("Boxy")
  {NumberOfSides = 4, SideLength = 4};
sq.Draw();
// Следующий код не скомпилируется:
// Console.WriteLine($"{sq.PetName} has {sq.NumberOfSides} of length
{sq.SideLength} and a
perimeter of {sq.Perimeter}");
```

Взамен экземпляр Square потребуется явно привести к интерфейсу IRegularPointy (т.к. реализация находится именно там) и тогда можно будет получать доступ к свойству Perimeter. Модифицируйте код следующим образом:

```
Console.WriteLine($"{sq.PetName} has {sq.NumberOfSides} of length
{sq.SideLength} and a perimeter of {((IRegularPointy)sq).Perimeter}");
```

Один из способов обхода этой проблемы — всегда указывать интерфейс типа. Измените определение экземпляра `Square`, указав вместо типа `Square` тип `IRegularPointy`:

```
IRegularPointy sq = new Square("Boxy") {NumberOfSides = 4, SideLength = 4};
```

Проблема с таким подходом (в данном случае) связана с тем, что метод `Draw()` и свойство `PetName` в интерфейсе не определены, а потому на этапе компиляции возникнут ошибки.

Хотя пример тривиален, он демонстрирует одну из проблем, касающихся стандартных реализаций. Прежде чем задействовать это средство в своем коде, обязательно оцените последствия того, что вызывающему коду должно быть известно, где находятся реализации.

Статические конструкторы и члены (нововведение в версии 8.0)

Еще одним дополнением интерфейсов в C# 8.0 является возможность наличия в них статических конструкторов и членов, которые функционируют аналогично статическим членам в определениях классов, но определены в интерфейсах. Добавьте к интерфейсу `IRegularPointy` статическое свойство и статический конструктор:

```
interface IRegularPointy : IPointy
{
  int SideLength { get; set; }
  int NumberOfSides { get; set; }
  int Perimeter => SideLength * NumberOfSides;

  // Статические члены также разрешены в версии C# 8.
  static string ExampleProperty { get; set; }

  static IRegularPointy() => ExampleProperty = "Foo";
}
```

Статические конструкторы не должны иметь параметры и могут получать доступ только к статическим свойствам и методам. Для обращения к статическому свойству интерфейса добавьте к операторам верхнего уровня следующий код:

```
Console.WriteLine($"Example property: {IRegularPointy.ExampleProperty}");
IRegularPointy.ExampleProperty = "Updated";
Console.WriteLine($"Example property: {IRegularPointy.ExampleProperty}");
```

Обратите внимание, что к статическому свойству необходимо обращаться через интерфейс, а не переменную экземпляра.

Использование интерфейсов в качестве параметров

Учитывая, что интерфейсы являются допустимыми типами, можно строить методы, которые принимают интерфейсы в качестве параметров, как было проиллюстрировано на примере метода `CloneMe()` ранее в главе. Предположим, что вы определили в текущем примере еще один интерфейс по имени `IDraw3D`:

```
namespace CustomInterfaces
{
  // Моделирует способность визуализации типа в трехмерном виде.
  public interface IDraw3D
  {
    void Draw3D();
  }
}
```

Далее сконфигурируйте две из трех фигур (`Circle` и `Hexagon`) с целью поддержки нового поведения:

```
// Circle поддерживает IDraw3D.
class ThreeDCircle : Circle, IDraw3D
{
  ...
  public void Draw3D()
    => Console.WriteLine("Drawing Circle in 3D!"); }
}
// Hexagon поддерживает IPointy и IDraw3D.
class Hexagon : Shape, IPointy, IDraw3D
{
  ...
  public void Draw3D()
    => Console.WriteLine("Drawing Hexagon in 3D!");
}
```

На рис. 8.2 показана обновленная диаграмма классов в Visual Studio.

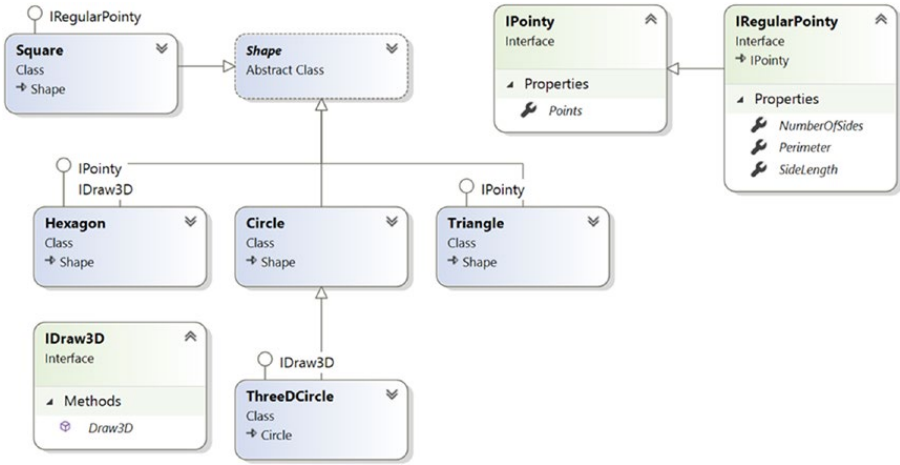

Рис. 8.2. Обновленная иерархия фигур

Теперь если вы определите метод, принимающий интерфейс `IDraw3D` в качестве параметра, тогда ему можно будет передавать по существу любой объект, реализующий `IDraw3D`. Попытка передачи типа, не поддерживающего необходимый интер-

фейс, приводит ошибке на этапе компиляции. Взгляните на следующий метод, определенный в классе Program:

```
// Будет рисовать любую фигуру, поддерживающую IDraw3D.
static void DrawIn3D(IDraw3D itf3d)
{
  Console.WriteLine("-> Drawing IDraw3D compatible type");
  itf3d.Draw3D();
}
```

Далее вы можете проверить, поддерживает ли элемент в массиве Shape новый интерфейс, и если поддерживает, то передать его методу DrawIn3D() на обработку:

```
Console.WriteLine("***** Fun with Interfaces *****\n");
Shape[] myShapes = { new Hexagon(), new Circle(),
  new Triangle("Joe"), new Circle("JoJo") } ;
for(int i = 0; i < myShapes.Length; i++)
{
  // Можно ли нарисовать эту фигуру в трехмерном виде?
  if (myShapes[i] is IDraw3D s)
  {
    DrawIn3D(s);
  }
}
```

Ниже представлен вывод, полученный из модифицированной версии приложения. Обратите внимание, что в трехмерном виде отображается только объект Hexagon, т.к. все остальные члены массива Shape не реализуют интерфейс IDraw3D:

```
***** Fun with Interfaces *****
...
-> Drawing IDraw3D compatible type
Drawing Hexagon in 3D!
```

Использование интерфейсов в качестве возвращаемых значений

Интерфейсы могут также применяться в качестве типов возвращаемых значений методов. Например, можно было бы написать метод, который получает массив объектов Shape и возвращает ссылку на первый элемент, поддерживающий IPointy:

```
// Этот метод возвращает первый объект в массиве,
// который реализует интерфейс IPointy.
static IPointy FindFirstPointyShape(Shape[] shapes)
{
  foreach (Shape s in shapes)
  {
    if (s is IPointy ip)
    {
      return ip;
    }
  }
  return null;
}
```

Взаимодействовать с методом FindFirstPointyShape() можно так:

```
Console.WriteLine("***** Fun with Interfaces *****\n");
// Создать массив элементов Shape.
Shape[] myShapes = { new Hexagon(), new Circle(),
                     new Triangle("Joe"), new Circle("JoJo")};
// Получить первый элемент, имеющий вершины.
IPointy firstPointyItem = FindFirstPointyShape(myShapes);
// В целях безопасности использовать null-условную операцию.
Console.WriteLine("The item has {0} points",
  firstPointyItem?.Points);
```

Массивы интерфейсных типов

Вспомните, что один интерфейс может быть реализован множеством типов, даже если они не находятся внутри той же самой иерархии классов и не имеют общего родительского класса помимо System.Object. Это позволяет формировать очень мощные программные конструкции. Например, пусть в текущем проекте разработаны три новых класса: два класса (Knife (нож) и Fork (вилка)) моделируют кухонные приборы, а третий (PitchFork (вилы)) — садовый инструмент. Ниже показан соответствующий код, а на рис. 8.3 — обновленная диаграмма классов.

```
// Fork.cs
namespace CustomInterfaces
{
  class Fork : IPointy
  {
    public byte Points => 4;
  }
}

// PitchFork.cs
namespace CustomInterfaces
{
  class PitchFork : IPointy
  {
    public byte Points => 3;
  }
}

// Knife.cs
namespace CustomInterfaces
{
  class Knife : IPointy
  {
    public byte Points => 1;
  }
}
```

После определения типов PitchFork, Fork и Knife можно определить массив объектов, совместимых с IPointy. Поскольку все элементы поддерживают один и тот же интерфейс, допускается выполнять проход по массиву и интерпретировать каждый его элемент как объект, совместимый с IPointy, несмотря на разнородность иерархий классов:

```
...
// Этот массив может содержать только типы,
// которые реализуют интерфейс IPointy.
IPointy[] myPointyObjects = {new Hexagon(), new Knife(),
  new Triangle(), new Fork(), new PitchFork()};
foreach(IPointy i in myPointyObjects)
{
  Console.WriteLine("Object has {0} points.", i.Points);
}
Console.ReadLine();
```

Просто чтобы подчеркнуть важность продемонстрированного примера, запомните, что массив заданного интерфейсного типа может содержать элементы любых классов или структур, реализующих этот интерфейс.

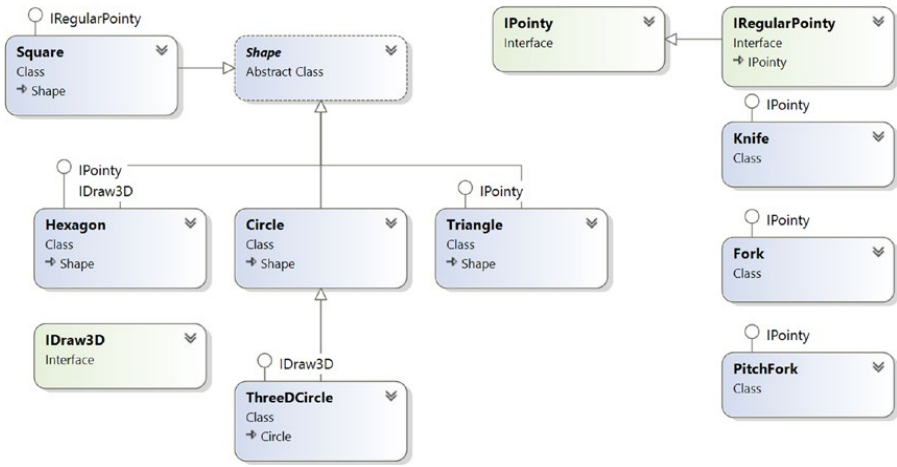

Рис. 8.3. Вспомните, что интерфейсы могут "подключаться" к любому типу внутри любой части иерархии классов

Автоматическая реализация интерфейсов

Хотя программирование на основе интерфейсов является мощным приемом, реализация интерфейсов может быть сопряжена с довольно большим объемом клавиатурного ввода. Учитывая, что интерфейсы являются именованными наборами абстрактных членов, для *каждого* метода интерфейса в *каждом* типе, который поддерживает данное поведение, потребуется вводить определение и реализацию. Следовательно, если вы хотите поддерживать интерфейс, который определяет пять методов и три свойства, тогда придется принять во внимание все восемь членов (иначе возникнут ошибки на этапе компиляции).

К счастью, в Visual Studio и Visual Studio Code поддерживаются разнообразные инструменты, упрощающие задачу реализации интерфейсов. В качестве примера вставьте в текущий проект еще один класс по имени `PointyTestClass`. Когда вы добавите к типу класса интерфейс, такой как `IPointy` (или любой другой подходящий интерфейс), то заметите, что по окончании ввода имени интерфейса (или при наведе-

нии на него курсора мыши в окне редактора кода) в Visual Studio и Visual Studio Code появляется значок с изображением лампочки (его также можно отобразить с помощью комбинации клавиш <Ctrl+.>). Щелчок на значке с изображением лампочки приводит к отображению раскрывающегося списка, который позволяет реализовать интерфейс (рис. 8.4 и 8.5).

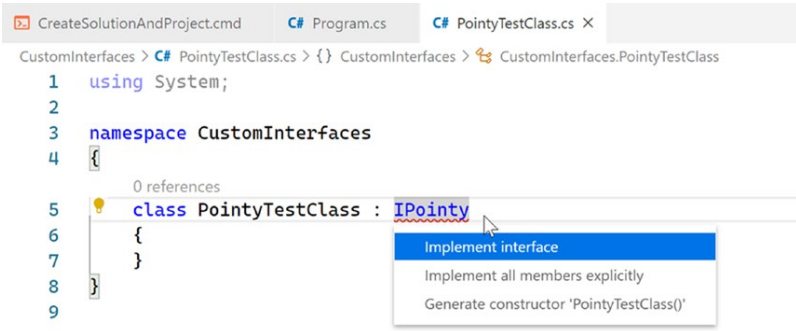

Рис. 8.4. Автоматическая реализация интерфейсов в Visual Studio Code

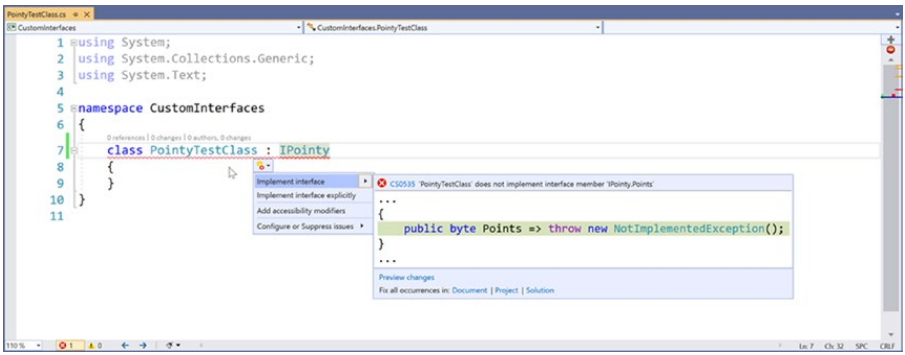

Рис. 8.5. Автоматическая реализация интерфейсов в Visual Studio

Обратите внимание, что в списке предлагаются два пункта, из которых второй (явная реализация интерфейса) обсуждается в следующем разделе. Для начала выберите первый пункт. Среда Visual Studio/Visual Studio Code сгенерирует код заглушки, подлежащий обновлению (как видите, стандартная реализация генерирует исключение `System.NotImplementedException`, что вполне очевидно можно удалить):

```
namespace CustomInterfaces
{
  class PointyTestClass : IPointy
  {
    public byte Points => throw new NotImplementedException();
  }
}
```

На заметку! Среда Visual Studio/Visual Studio Code также поддерживает рефакторинг в форме извлечения интерфейса (Extract Interface), доступный через пункт Extract Interface (Извлечь интерфейс) меню Quick Actions (Быстрые действия). Такой рефакторинг позволяет извлечь новое определение интерфейса из существующего определения класса. Например, вы можете находиться где-то на полпути к завершению написания класса, но вдруг осознаете, что данное поведение можно обобщить в виде интерфейса (открывая возможность для альтернативных реализаций).

Явная реализация интерфейсов

Как было показано ранее в главе, класс или структура может реализовывать любое количество интерфейсов. С учетом этого всегда существует возможность реализации интерфейсов, которые содержат члены с идентичными именами, из-за чего придется устранять конфликты имен. Чтобы проиллюстрировать разнообразные способы решения данной проблемы, создайте новый проект консольного приложения по имени `InterfaceNameClash` и добавьте в него три специальных интерфейса, представляющих различные места, в которых реализующий их тип может визуализировать свой вывод:

```
namespace InterfaceNameClash
{
  // Вывести изображение на форму.
  public interface IDrawToForm
  {
    void Draw();
  }
}

namespace InterfaceNameClash
{
  // Вывести изображение в буфер памяти.
  public interface IDrawToMemory
  {
    void Draw();
  }
}

namespace InterfaceNameClash
{
  // Вывести изображение на принтер.
  public interface IDrawToPrinter
  {
    void Draw();
  }
}
```

Обратите внимание, что в каждом интерфейсе определен метод по имени `Draw()` с идентичной сигнатурой. Если все объявленные интерфейсы необходимо поддерживать в одном классе `Octagon`, то компилятор разрешит следующее определение:

```
using System;
namespace InterfaceNameClash
{
```

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  public void Draw()
  {
    // Разделяемая логика вывода.
    Console.WriteLine("Drawing the Octagon...");
  }
}
```

Хотя компиляция такого кода пройдет гладко, здесь присутствует потенциальная проблема. Выражаясь просто, предоставление единственной реализации метода `Draw()` не позволяет предпринимать уникальные действия на основе того, какой интерфейс получен от объекта `Octagon`. Например, представленный ниже код будет приводить к вызову того же самого метода `Draw()` независимо от того, какой интерфейс получен:

```
using System;
using InterfaceNameClash;

Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
// Все эти обращения приводят к вызову одного
// и того же метода Draw()!
Octagon oct = new Octagon();

// Сокращенная форма записи, если переменная типа
// интерфейса в дальнейшем использоваться не будет.
((IDrawToPrinter)oct).Draw();

// Также можно применять ключевое слово is.
if (oct is IDrawToMemory dtm)
{
  dtm.Draw();
}
```

Очевидно, что код, требуемый для визуализации изображения в окне, значительно отличается от кода, который необходим для вывода изображения на сетевой принтер или в область памяти. При реализации нескольких интерфейсов, имеющих идентичные члены, разрешить подобный конфликт имен можно с применением синтаксиса *явной реализации интерфейсов*. Взгляните на следующую модификацию типа `Octagon`:

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  // Явно привязать реализации Draw() к конкретным интерфейсам.
  void IDrawToForm.Draw()
  {
    Console.WriteLine("Drawing to form...");       // Вывод на форму
  }
  void IDrawToMemory.Draw()
  {
    Console.WriteLine("Drawing to memory...");     // Вывод в память
  }
  void IDrawToPrinter.Draw()
  {
    Console.WriteLine("Drawing to a printer...");  // Вывод на принтер
  }
}
```

Как видите, при явной реализации члена интерфейса общий шаблон выглядит следующим образом:

```
возвращаемыйТип ИмяИнтерфейса.ИмяМетода(параметры) {}
```

Обратите внимание, что при использовании такого синтаксиса модификатор доступа не указывается; явно реализованные члены автоматически будут закрытыми. Например, такой синтаксис недопустим:

```
// Ошибка! Модификатор доступа не может быть указан!
public void IDrawToForm.Draw()
{
  Console.WriteLine("Drawing to form...");
}
```

Поскольку явно реализованные члены всегда неявно закрыты, они перестают быть доступными на уровне объектов. Фактически, если вы примените к типу `Octagon` операцию точки, то обнаружите, что средство IntelliSense не отображает члены `Draw()`. Как и следовало ожидать, для доступа к требуемой функциональности должно использоваться явное приведение. В предыдущих операторах верхнего уровня уже используется явное приведение, так что они работают с явными интерфейсами.

```
Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
Octagon oct = new Octagon();

// Теперь для доступа к членам Draw() должно
// использоваться приведение.
IDrawToForm itfForm = (IDrawToForm)oct;
itfForm.Draw();

// Сокращенная форма записи, если переменная типа
// интерфейса в дальнейшем использоваться не будет.
((IDrawToPrinter)oct).Draw();

// Также можно применять ключевое слово is.
if (oct is IDrawToMemory dtm)
{
  dtm.Draw();
}
Console.ReadLine();
```

Наряду с тем, что этот синтаксис действительно полезен, когда необходимо устранить конфликты имен, явную реализацию интерфейсов можно применять и просто для сокрытия более "сложных" членов на уровне объектов. В таком случае при использовании операции точки пользователь объекта будет видеть только подмножество всей функциональности типа. Однако когда требуется более сложное поведение, желаемый интерфейс можно извлекать через явное приведение.

Проектирование иерархий интерфейсов

Интерфейсы могут быть организованы в иерархии. Подобно иерархии классов, когда интерфейс расширяет существующий интерфейс, он наследует все абстрактные члены, определяемые родителем (или родителями). До выхода версии C# 8 производный интерфейс не наследовал действительную реализацию, а просто расширял собственное определение дополнительными абстрактными членами. В версии C# 8 производные интерфейсы наследуют стандартные реализации, а также расширяют свои определения и потенциально добавляют новые стандартные реализации.

Иерархии интерфейсов могут быть удобными, когда нужно расширить функциональность имеющегося интерфейса, не нарушая работу существующих кодовых баз. В целях иллюстрации создайте новый проект консольного приложения по имени `InterfaceHierarchy`. Затем спроектируйте новый набор интерфейсов, связанных с визуализацией, таким образом, чтобы `IDrawable` был корневым интерфейсом в дереве семейства:

```
namespace InterfaceHierarchy
{
  public interface IDrawable
  {
    void Draw();
  }
}
```

С учетом того, что интерфейс `IDrawable` определяет базовое поведение рисования, можно создать производный интерфейс, который расширяет `IDrawable` возможностью визуализации в других форматах, например:

```
namespace InterfaceHierarchy
{
  public interface IAdvancedDraw : IDrawable
  {
    void DrawInBoundingBox(int top, int left, int bottom, int right);
    void DrawUpsideDown();
  }
}
```

При таком проектном решении, если класс реализует интерфейс `IAdvancedDraw`, тогда ему потребуется реализовать все члены, определенные в цепочке наследования (в частности методы `Draw()`, `DrawInBoundingBox()` и `DrawUpsideDown()`):

```
using System;
namespace InterfaceHierarchy
{
  public class BitmapImage : IAdvancedDraw
  {
    public void Draw()
    {
      Console.WriteLine("Drawing...");
    }
    public void DrawInBoundingBox(int top, int left, int bottom,
                                  int right)
    {
      Console.WriteLine("Drawing in a box...");
    }
    public void DrawUpsideDown()
    {
      Console.WriteLine("Drawing upside down!");
    }
  }
}
```

Теперь в случае применения класса BitmapImage появилась возможность вызывать каждый метод на уровне объекта (из-за того, что все они открыты), а также извлекать ссылку на каждый поддерживаемый интерфейс явным образом через приведение:

```
using System;
using InterfaceHierarchy;

Console.WriteLine("***** Simple Interface Hierarchy *****");

// Вызвать на уровне объекта.
BitmapImage myBitmap = new BitmapImage();
myBitmap.Draw();
myBitmap.DrawInBoundingBox(10, 10, 100, 150);
myBitmap.DrawUpsideDown();

// Получить IAdvancedDraw явным образом.
if (myBitmap is IAdvancedDraw iAdvDraw)
{
  iAdvDraw.DrawUpsideDown();
}
Console.ReadLine();
```

Иерархии интерфейсов со стандартными реализациями (нововведение в версии 8.0)

Когда иерархии интерфейсов также включают стандартные реализации, то нижерасположенные интерфейсы могут задействовать реализацию из базового интерфейса или создать новую стандартную реализацию. Модифицируйте интерфейс IDrawable, как показано ниже:

```
public interface IDrawable
{
  void Draw();
  int TimeToDraw() => 5;
}
```

Теперь обновите операторы верхнего уровня:

```
Console.WriteLine("***** Simple Interface Hierarchy *****");
...
if (myBitmap is IAdvancedDraw iAdvDraw)
{
  iAdvDraw.DrawUpsideDown();
  Console.WriteLine($"Time to draw: {iAdvDraw.TimeToDraw()}");
}
Console.ReadLine();
```

Этот код не только скомпилируется, но и выведет значение 5 при вызове метода TimeToDraw(). Дело в том, что стандартные реализации попадают в производные интерфейсы автоматически. Приведение BitMapImage к интерфейсу IAdvancedDraw обеспечивает доступ к методу TimeToDraw(), хотя экземпляр BitMapImage не имеет доступа к стандартной реализации. Чтобы удостовериться в этом, введите следующий код, который вызовет ошибку на этапе компиляции:

```
// Этот код не скомпилируется.
myBitmap.TimeToDraw();
```

Если в нижерасположенном интерфейсе желательно предоставить собственную стандартную реализацию, тогда потребуется скрыть вышерасположенную реализацию. Например, если вычерчивание в методе `TimeToDraw()` из `IAdvancedDraw` занимает 15 единиц времени, то модифицируйте определение интерфейса следующим образом:

```
public interface IAdvancedDraw : IDrawable
{
  void DrawInBoundingBox(int top, int left, int bottom, int right);
  void DrawUpsideDown();
  new int TimeToDraw() => 15;
}
```

Разумеется, в классе `BitMapImage` также можно реализовать метод `TimeToDraw()`. В отличие от метода `TimeToDraw()` из `IAdvancedDraw` в классе необходимо только реализовать метод без его сокрытия.

```
public class BitmapImage : IAdvancedDraw
{
  ...
  public int TimeToDraw() => 12;
}
```

В случае приведения экземпляра `BitmapImage` к интерфейсу `IAdvancedDraw` или `IDrawable` метод на экземпляре по-прежнему выполняется. Добавьте к операторам верхнего уровня показанный далее код:

```
// Всегда вызывается метод на экземпляре:
Console.WriteLine("***** Calling Implemented TimeToDraw *****");
Console.WriteLine($"Time to draw: {myBitmap.TimeToDraw()}");
Console.WriteLine($"Time to draw: {((IDrawable) myBitmap).TimeToDraw()}");
Console.WriteLine($"Time to draw: {((IAdvancedDraw) myBitmap).TimeToDraw()}");
```

Вот результаты:

```
***** Simple Interface Hierarchy *****
...
***** Calling Implemented TimeToDraw *****
Time to draw: 12
Time to draw: 12
Time to draw: 12
```

Множественное наследование с помощью интерфейсных типов

В отличие от типов классов интерфейс может расширять множество базовых интерфейсов, что позволяет проектировать мощные и гибкие абстракции. Создайте новый проект консольного приложения по имени `MiInterfaceHierarchy`. Здесь имеется еще одна коллекция интерфейсов, которые моделируют разнообразные абстракции, связанные с визуализацией и фигурами. Обратите внимание, что интерфейс `IShape` расширяет и `IDrawable`, и `IPrintable`:

```
// IDrawable.cs
namespace MiInterfaceHierarchy
{
  // Множественное наследование для интерфейсных типов - это нормально.
  interface IDrawable
  {
    void Draw();
  }
}
```

```
// IPrintable.cs
namespace MiInterfaceHierarchy
{
  interface IPrintable
  {
    void Print();
    void Draw();  // <-- Здесь возможен конфликт имен!
  }
}
// IShape.cs
namespace MiInterfaceHierarchy
{
  // Множественное наследование интерфейсов. Нормально!
  interface IShape : IDrawable, IPrintable
  {
    int GetNumberOfSides();
  }
}
```

На рис. 8.6 показана текущая иерархия интерфейсов.

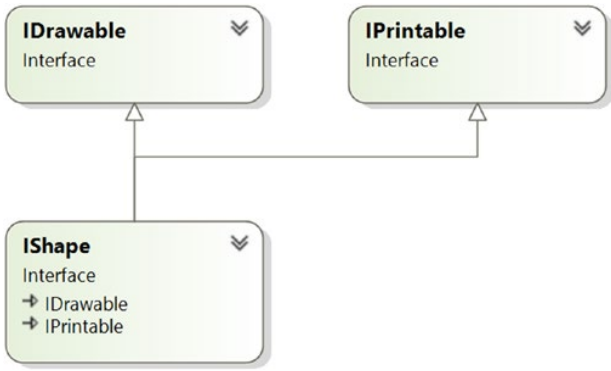

Рис. 8.6. В отличие от классов интерфейсы могут расширять сразу несколько базовых интерфейсов

Главный вопрос: сколько методов должен реализовывать класс, поддерживающий `IShape`? Ответ: в зависимости от обстоятельств. Если вы хотите предоставить простую реализацию метода `Draw()`, тогда вам необходимо реализовать только три члена, как иллюстрируется в следующем типе `Rectangle`:

```
using System;

namespace MiInterfaceHierarchy
{
  class Rectangle : IShape
  {
    public int GetNumberOfSides() => 4;
    public void Draw() => Console.WriteLine("Drawing...");
    public void Print() => Console.WriteLine("Printing...");
  }
}
```

Если вы предпочитаете располагать специфическими реализациями для каждого метода `Draw()` (что в данном случае имеет смысл), то конфликт имен можно устранить с использованием явной реализации интерфейсов, как делается в представленном далее типе `Square`:

```
namespace MiInterfaceHierarchy
{
  class Square : IShape
  {
    // Использование явной реализации для устранения
    // конфликта имен членов.
    void IPrintable.Draw()
    {
      // Вывести на принтер...
    }
    void IDrawable.Draw()
    {
      // Вывести на экран...
    }
    public void Print()
    {
      // Печатать...
    }
    public int GetNumberOfSides() => 4;
  }
}
```

В идеале к данному моменту вы должны лучше понимать процесс определения и реализации специальных интерфейсов с применением синтаксиса C#. По правде говоря, привыкание к программированию на основе интерфейсов может занять определенное время, так что если вы находитесь в некотором замешательстве, то это совершенно нормальная реакция.

Однако имейте в виду, что интерфейсы являются фундаментальным аспектом .NET Core. Независимо от типа разрабатываемого приложения (веб-приложение, настольное приложение с графическим пользовательским интерфейсом, библиотека доступа к данным и т.п.) работа с интерфейсами будет составной частью этого процесса. Подводя итог, запомните, что интерфейсы могут быть исключительно полезны в следующих ситуациях:

- существует единственная иерархия, в которой общее поведение поддерживается только подмножеством производных типов;
- необходимо моделировать общее поведение, которое встречается в нескольких иерархиях, не имеющих общего родительского класса кроме `System.Object`.

Итак, вы ознакомились со спецификой построения и реализации специальных интерфейсов. Остаток главы посвящен исследованию нескольких предопределенных интерфейсов, содержащихся в библиотеках базовых классов .NET Core. Как будет показано, вы можете реализовывать стандартные интерфейсы .NET Core в своих специальных типах, обеспечивая их бесшовную интеграцию с инфраструктурой.

Интерфейсы IEnumerable и IEnumerator

Прежде чем приступать к исследованию процесса реализации существующих интерфейсов .NET Core, давайте сначала рассмотрим роль интерфейсов `IEnumerable` и `IEnumerator`. Вспомните, что язык C# поддерживает ключевое слово `foreach`, которое позволяет осуществлять проход по содержимому массива любого типа:

```
// Итерация по массиву элементов.
int[] myArrayOfInts = {10, 20, 30, 40};

foreach(int i in myArrayOfInts)
{
  Console.WriteLine(i);
}
```

Хотя может показаться, что данная конструкция подходит только для массивов, на самом деле `foreach` разрешено использовать с любым типом, который поддерживает метод `GetEnumerator()`. В целях иллюстрации создайте новый проект консольного приложения по имени `CustomEnumerator`. Скопируйте в новый проект файлы `Car.cs` и `Radio.cs` из проекта `SimpleException`, рассмотренного в главе 7. Не забудьте поменять пространства имен для классов на `CustomEnumerator`.

Теперь вставьте в проект новый класс `Garage` (гараж), который хранит набор объектов `Car` (автомобиль) внутри `System.Array`:

```
using System.Collections;

namespace CustomEnumerator
{
  // Garage содержит набор объектов Car.
  public class Garage
  {
    private Car[] carArray = new Car[4];

    // При запуске заполнить несколькими объектами Car.
    public Garage()
    {
      carArray[0] = new Car("Rusty", 30);
      carArray[1] = new Car("Clunker", 55);
      carArray[2] = new Car("Zippy", 30);
      carArray[3] = new Car("Fred", 30);
    }
  }
}
```

В идеальном случае было бы удобно проходить по внутренним элементам объекта `Garage` с применением конструкции `foreach` как в ситуации с массивом значений данных:

```
using System;
using CustomEnumerator;

// Код выглядит корректным...
Console.WriteLine("***** Fun with IEnumerable / IEnumerator *****\n");
Garage carLot = new Garage();

// Проход по всем объектам Car в коллекции?
foreach (Car c in carLot)
```

```
{
  Console.WriteLine("{0} is going {1} MPH",
    c.PetName, c.CurrentSpeed);
}
Console.ReadLine();
```

К сожалению, компилятор информирует о том, что в классе `Garage` не реализован метод по имени `GetEnumerator()`, который формально определен в интерфейсе `IEnumerable`, находящемся в пространстве имен `System.Collections`.

На заметку! В главе 10 вы узнаете о роли обобщений и о пространстве имен `System.Collections.Generic`. Как будет показано, это пространство имен содержит обобщенные версии интерфейсов `IEnumerable/IEnumerator`, которые предлагают более безопасный к типам способ итерации по элементам.

Классы или структуры, которые поддерживают такое поведение, позиционируются как способные предоставлять вызывающему коду доступ к элементам, содержащимся внутри них (в рассматриваемом примере самому ключевому слову `foreach`). Вот определение этого стандартного интерфейса:

```
// Данный интерфейс информирует вызывающий код о том,
// что элементы объекта могут перечисляться.
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Как видите, метод `GetEnumerator()` возвращает ссылку на еще один интерфейс по имени `System.Collections.IEnumerator`, обеспечивающий инфраструктуру, которая позволяет вызывающему коду обходить внутренние объекты, содержащиеся в совместимом с `IEnumerable` контейнере:

```
// Этот интерфейс позволяет вызывающему коду получать элементы контейнера.
public interface IEnumerator
{
  bool MoveNext (); // Переместить вперед внутреннюю позицию курсора.
  object Current { get; }  // Получить текущий элемент
                           // (свойство только для чтения).
  void Reset ();    // Сбросить курсор в позицию перед первым элементом.
}
```

Если вы хотите обновить тип `Garage` для поддержки этих интерфейсов, то можете пойти длинным путем и реализовать каждый метод вручную. Хотя вы определенно вольны предоставить специализированные версии методов `GetEnumerator()`, `MoveNext()`, `Current` и `Reset()`, существует более легкий путь. Поскольку тип `System.Array` (а также многие другие классы коллекций) уже реализует интерфейсы `IEnumerable` и `IEnumerator`, вы можете просто делегировать запрос к `System.Array` следующим образом (обратите внимание, что в файл кода понадобится импортировать пространство имен `System.Collections`):

```
using System.Collections;
...
public class Garage : IEnumerable
{
```

```
  // System.Array уже реализует IEnumerator!
  private Car[] carArray = new Car[4];

  public Garage()
  {
    carArray[0] = new Car("FeeFee", 200);
    carArray[1] = new Car("Clunker", 90);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
  }

  // Возвратить IEnumerator объекта массива.
  public IEnumerator GetEnumerator()
    => carArray.GetEnumerator();
}
```

После такого изменения тип `Garage` можно безопасно использовать внутри конструкции `foreach`. Более того, учитывая, что метод `GetEnumerator()` был определен как открытый, пользователь объекта может также взаимодействовать с типом `IEnumerator`:

```
// Вручную работать с IEnumerator.
IEnumerator carEnumerator = carLot.GetEnumerator();
carEnumerator.MoveNext();
Car myCar = (Car)carEnumerator.Current;
Console.WriteLine("{0} is going {1} MPH", myCar.PetName, myCar.CurrentSpeed);
```

Тем не менее, если вы предпочитаете скрыть функциональность `IEnumerable` на уровне объектов, то просто задействуйте явную реализацию интерфейса:

```
// Возвратить IEnumerator объекта массива.
IEnumerator IEnumerable.GetEnumerator()
  => return carArray.GetEnumerator();
```

В результате обычный пользователь объекта не обнаружит метод `GetEnumerator()` в классе `Garage`, в то время как конструкция `foreach` при необходимости будет получать интерфейс в фоновом режиме.

Построение итераторных методов с использованием ключевого слова `yield`

Существует альтернативный способ построения типов, которые работают с циклом `foreach`, предусматривающий использование *итераторов*. Попросту говоря, *итератор* — это член, который указывает, каким образом должны возвращаться внутренние элементы контейнера во время обработки в цикле `foreach`. В целях иллюстрации создайте новый проект консольного приложения по имени `CustomEnumeratorWithYield` и вставьте в него типы `Car`, `Radio` и `Garage` из предыдущего примера (снова переименовав пространство имен согласно текущему проекту). Затем модифицируйте тип `Garage`:

```
public class Garage : IEnumerable
{
  ...
  // Итераторный метод.
  public IEnumerator GetEnumerator()
  {
```

```
    foreach (Car c in carArray)
    {
      yield return c;
    }
  }
}
```

Обратите внимание, что показанная реализация метода `GetEnumerator()` осуществляет проход по элементам с применением внутренней логики `foreach` и возвращает каждый объект `Car` вызывающему коду, используя синтаксис `yield return`. Ключевое слово `yield` применяется для указания значения или значений, которые подлежат возвращению конструкцией `foreach` вызывающему коду. При достижении оператора `yield return` текущее местоположение в контейнере сохраняется и выполнение возобновляется с этого местоположения, когда итератор вызывается в следующий раз.

Итераторные методы не обязаны использовать ключевое слово `foreach` для возвращения своего содержимого. Итераторный метод допускается определять и так:

```
public IEnumerator GetEnumerator()
{
  yield return carArray[0];
  yield return carArray[1];
  yield return carArray[2];
  yield return carArray[3];
}
```

В этой реализации обратите внимание на то, что при каждом своем прохождении метод `GetEnumerator()` явно возвращает вызывающему коду новое значение. В рассматриваемом примере поступать подобным образом мало смысла, потому что если вы добавите дополнительные объекты к переменной-члену `carArray`, то метод `GetEnumerator()` станет рассогласованным. Тем не менее, такой синтаксис может быть полезен, когда вы хотите возвращать из метода локальные данные для обработки посредством `foreach`.

Защитные конструкции с использованием локальных функций (нововведение в версии 7.0)

До первого прохода по элементам (или доступа к любому элементу) никакой код в методе `GetEnumerator()` не выполняется. Таким образом, если до выполнения оператора `yield` возникает условие для исключения, то оно не будет сгенерировано при первом вызове метода, а лишь во время первого вызова `MoveNext()`.

Чтобы проверить это, модифицируйте `GetEnumerator()`:

```
public IEnumerator GetEnumerator()
{
  // Исключение не сгенерируется до тех пор, пока не будет вызван
  // метод MoveNext().
  throw new Exception("This won't get called");
  foreach (Car c in carArray)
  {
    yield return c;
  }
}
```

Если функция вызывается, как показано далее, и *больше ничего не делается*, тогда исключение никогда не сгенерируется:

```
using System.Collections;
...
Console.WriteLine("***** Fun with the Yield Keyword *****\n");
Garage carLot = new Garage();
IEnumerator carEnumerator = carLot.GetEnumerator();

Console.ReadLine();
```

Код выполнится только после вызова `MoveNext()` и сгенерируется исключение. В зависимости от нужд программы это может быть как вполне нормально, так и нет. Ваш метод `GetEnumerator()` может иметь *защитную конструкцию*, которую необходимо выполнить при вызове метода в первый раз. В качестве примера предположим, что список формируется из базы данных. Вам может понадобиться организовать проверку, открыто ли подключение к базе данных, во время *вызова* метода, а не при проходе по списку. Или же может возникнуть потребность в проверке достоверности входных параметров метода `Iterator()`, который рассматривается далее.

Вспомните средство локальных функций версии C# 7, представленное в главе 4; локальные функции — это закрытые функции, которые определены внутри других функций. За счет перемещения `yield return` внутрь локальной функции, которая возвращается из главного тела метода, операторы верхнего уровня (до возвращения локальной функции) выполняются немедленно. Локальная функция выполняется при вызове `MoveNext()`.

Приведите метод к следующему виду:

```
public IEnumerator GetEnumerator()
{
  // Это исключение сгенерируется немедленно.
  throw new Exception("This will get called");

  return ActualImplementation();

  // Локальная функция и фактическая реализация IEnumerator.
  IEnumerator ActualImplementation()
  {
    foreach (Car c in carArray)
    {
      yield return c;
    }
  }
}
```

Ниже показан тестовый код:

```
Console.WriteLine("***** Fun with the Yield Keyword *****\n");
Garage carLot = new Garage();
try
{
  // На этот раз возникает ошибка.
  var carEnumerator = carLot.GetEnumerator();
}
catch (Exception e)
{
  Console.WriteLine($"Exception occurred on GetEnumerator");
}
Console.ReadLine();
```

В результате такого обновления метода `GetEnumerator()` исключение генерируется незамедлительно, а не при вызове `MoveNext()`.

Построение именованного итератора

Также интересно отметить, что ключевое слово `yield` формально может применяться внутри любого метода независимо от его имени. Такие методы (которые официально называются *именованными итераторами*) уникальны тем, что способны принимать любое количество аргументов. При построении именованного итератора имейте в виду, что метод будет возвращать интерфейс `IEnumerable`, а не ожидаемый совместимый с `IEnumerator` тип. В целях иллюстрации добавьте к типу `Garage` следующий метод (использующий локальную функцию для инкапсуляции функциональности итерации):

```
public IEnumerable GetTheCars(bool returnReversed)
{
  // Выполнить проверку на предмет ошибок.
  return ActualImplementation();

  IEnumerable ActualImplementation()
  {
    // Возвратить элементы в обратном порядке.
    if (returnReversed)
    {
      for (int i = carArray.Length; i != 0; i--)
      {
        yield return carArray[i - 1];
      }
    }
    else
    {
      // Возвратить элементы в том порядке, в каком они размещены в массиве.
      foreach (Car c in carArray)
      {
        yield return c;
      }
    }
  }
}
```

Обратите внимание, что новый метод позволяет вызывающему коду получать элементы в прямом, а также в обратном порядке, если во входном параметре указано значение `true`. Теперь взаимодействовать с методом `GetTheCars()` можно так (обязательно закомментируйте оператор `throw new` в методе `GetEnumerator()`):

```
Console.WriteLine("***** Fun with the Yield Keyword *****\n");
Garage carLot = new Garage();

// Получить элементы, используя GetEnumerator().
foreach (Car c in carLot)
{
  Console.WriteLine("{0} is going {1} MPH",
    c.PetName, c.CurrentSpeed);
}

Console.WriteLine();
```

```
// Получить элементы (в обратном порядке!)
// с применением именованного итератора.
foreach (Car c in carLot.GetTheCars(true))
{
  Console.WriteLine("{0} is going {1} MPH",
    c.PetName, c.CurrentSpeed);
}
Console.ReadLine();
```

Наверняка вы согласитесь с тем, что именованные итераторы являются удобными конструкциями, поскольку они позволяют определять в единственном специальном контейнере множество способов запрашивания возвращаемого набора.

Итак, в завершение темы построения перечислимых объектов запомните: для того, чтобы специальные типы могли работать с ключевым словом `foreach` языка C#, контейнер должен определять метод по имени `GetEnumerator()`, который формально определен интерфейсным типом `IEnumerable`. Этот метод обычно реализуется просто за счет делегирования работы внутреннему члену, который хранит подобъекты, но допускается также использовать синтаксис `yield return`, чтобы предоставить множество методов "именованных итераторов".

Интерфейс ICloneable

Вспомните из главы 6, что в классе `System.Object` определен метод по имени `MemberwiseClone()`, который применяется для получения *поверхностной (неглубокой) копии* текущего объекта. Пользователи объекта не могут вызывать указанный метод напрямую, т.к. он является защищенным. Тем не менее, отдельный объект может самостоятельно вызывать `MemberwiseClone()` во время процесса *клонирования*. В качестве примера создайте новый проект консольного приложения по имени `CloneablePoint`, в котором определен класс `Point`:

```
using System;

namespace CloneablePoint
{
  // Класс по имени Point.
  public class Point
  {
    public int X {get; set;}
    public int Y {get; set;}

    public Point(int xPos, int yPos) { X = xPos; Y = yPos;}
    public Point(){}

    // Переопределить Object.ToString().
    public override string ToString() => $"X = {X}; Y = {Y}";
  }
}
```

Учитывая имеющиеся у вас знания о ссылочных типах и типах значений (см. главу 4), должно быть понятно, что если вы присвоите одну переменную ссылочного типа другой такой переменной, то получите две ссылки, которые указывают на тот же самый объект в памяти. Таким образом, следующая операция присваивания в результате дает две ссылки на один и тот же объект `Point` в куче; модификация с использованием любой из ссылок оказывает воздействие на тот же самый объект в куче:

```
Console.WriteLine("***** Fun with Object Cloning *****\n");
// Две ссылки на один и тот же объект!
Point p1 = new Point(50, 50);
Point p2 = p1;
p2.X = 0;
Console.WriteLine(p1);
Console.WriteLine(p2);
Console.ReadLine();
```

Чтобы предоставить специальному типу возможность возвращения вызывающему коду идентичную копию самого себя, можно реализовать стандартный интерфейс `ICloneable`. Как было показано в начале главы, в интерфейсе `ICloneable` определен единственный метод по имени `Clone()`:

```
public interface ICloneable
{
  object Clone();
}
```

Очевидно, что реализация метода `Clone()` варьируется от класса к классу. Однако базовая функциональность в основном остается неизменной: копирование значений переменных-членов в новый объект того же самого типа и возвращение его пользователю. В целях демонстрации модифицируйте класс `Point`:

```
// Теперь Point поддерживает способность клонирования.
public class Point : ICloneable
{
  public int X { get; set; }
  public int Y { get; set; }

  public Point(int xPos, int yPos) { X = xPos; Y = yPos; }
  public Point() { }

  // Переопределить Object.ToString().
  public override string ToString() => $»X = {X}; Y = {Y}»;

  // Возвратить копию текущего объекта.
  public object Clone() => new Point(this.X, this.Y);
}
```

Теперь можно создавать точные автономные копии объектов типа `Point`:

```
Console.WriteLine("***** Fun with Object Cloning *****\n");
// Обратите внимание, что Clone() возвращает простой тип object.
// Для получения производного типа требуется явное приведение.
Point p3 = new Point(100, 100);
Point p4 = (Point)p3.Clone();

// Изменить p4.X (что не приводит к изменению p3.x).
p4.X = 0;

// Вывести все объекты.
Console.WriteLine(p3);
Console.WriteLine(p4);
Console.ReadLine();
```

Несмотря на то что текущая реализация типа `Point` удовлетворяет всем требованиям, есть возможность ее немного улучшить. Поскольку `Point` не содержит никаких внутренних переменных ссылочного типа, реализацию метода `Clone()` можно упростить:

```
// Копировать все поля Point по очереди.
public object Clone() => this.MemberwiseClone();
```

Тем не менее, учтите, что если бы в типе `Point` содержались любые переменные-члены ссылочного типа, то метод `MemberwiseClone()` копировал бы ссылки на эти объекты (т.е. создавал бы *поверхностную копию*). Для поддержки подлинной *глубокой (детальной) копии* во время процесса клонирования понадобится создавать новые экземпляры каждой переменной-члена ссылочного типа. Давайте рассмотрим пример.

Более сложный пример клонирования

Теперь предположим, что класс `Point` содержит переменную-член ссылочного типа `PointDescription`. Данный класс представляет дружественное имя точки, а также ее идентификационный номер, выраженный как `System.Guid` (глобально уникальный идентификатор (globally unique identifier — GUID), т.е. статистически уникальное 128-битное число). Вот как выглядит реализация:

```
using System;

namespace CloneablePoint
{
  // Этот класс описывает точку.
  public class PointDescription
  {
    public string PetName {get; set;}
    public Guid PointID {get; set;}

    public PointDescription()
    {
      PetName = "No-name";
      PointID = Guid.NewGuid();
    }
  }
}
```

Начальные изменения самого класса `Point` включают модификацию метода `ToString()` для учета новых данных состояния, а также определение и создание ссылочного типа `PointDescription`. Чтобы позволить внешнему миру устанавливать дружественное имя для `Point`, необходимо также изменить аргументы, передаваемые перегруженному конструктору:

```
public class Point : ICloneable
{
  public int X { get; set; }
  public int Y { get; set; }
  public PointDescription desc = new PointDescription();

  public Point(int xPos, int yPos, string petName)
  {
    X = xPos; Y = yPos;
    desc.PetName = petName;
  }
  public Point(int xPos, int yPos)
  {
    X = xPos; Y = yPos;
  }
```

```
  public Point() { }

  // Переопределить Object.ToString().
  public override string ToString()
    => $"X = {X}; Y = {Y}; Name = {desc.PetName};\nID = {desc.
PointID}\n";

  // Возвратить копию текущего объекта.
  public object Clone() => this.MemberwiseClone();
}
```

Обратите внимание, что метод `Clone()` пока еще не обновлялся. Следовательно, когда пользователь объекта запросит клонирование с применением текущей реализации, будет создана поверхностная (почленная) копия. В целях иллюстрации модифицируйте вызывающий код, как показано ниже:

```
Console.WriteLine("***** Fun with Object Cloning *****\n");
...
Console.WriteLine("Cloned p3 and stored new Point in p4");
Point p3 = new Point(100, 100, "Jane");
Point p4 = (Point)p3.Clone();

Console.WriteLine("Before modification:");  // Перед модификацией
Console.WriteLine("p3: {0}", p3);
Console.WriteLine("p4: {0}", p4);
p4.desc.PetName = "My new Point";
p4.X = 9;

Console.WriteLine("\nChanged p4.desc.petName and p4.X");
Console.WriteLine("After modification:");   // После модификации
Console.WriteLine("p3: {0}", p3);
Console.WriteLine("p4: {0}", p4);
Console.ReadLine();
```

В приведенном далее выводе видно, что хотя типы значений действительно были изменены, внутренние ссылочные типы поддерживают одни и те же значения, т.к. они "указывают" на те же самые объекты в памяти (в частности, оба объекта имеют дружественное имя `My new Point`):

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

Changed p4.desc.petName  and p4.X
After modification:
p3: X = 100; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 9; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
```

Чтобы заставить метод `Clone()` создавать полную глубокую копию внутренних ссылочных типов, нужно сконфигурировать объект, возвращаемый методом `MemberwiseClone()`, для учета имени текущего объекта `Point` (тип `System.Guid`

на самом деле является структурой, так что числовые данные будут действительно копироваться). Вот одна из возможных реализаций:

```
// Теперь необходимо скорректировать код для учета члена
PointDescription.
public object Clone()
{
  // Сначала получить поверхностную копию.
  Point newPoint = (Point)this.MemberwiseClone();

  // Затем восполнить пробелы.
  PointDescription currentDesc = new PointDescription();
  currentDesc.PetName = this.desc.PetName;
  newPoint.desc = currentDesc;
  return newPoint;
}
```

Если снова запустить приложение и просмотреть его вывод (показанный далее), то будет видно, что возвращаемый методом `Clone()` объект `Point` действительно копирует свои внутренние переменные-члены ссылочного типа (обратите внимание, что дружественные имена у `p3` и `p4` теперь уникальны):

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 100; Y = 100; Name = Jane;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a

Changed p4.desc.petName  and p4.X
After modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 9; Y = 100; Name = My new Point;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a
```

Давайте подведем итоги по процессу клонирования. При наличии класса или структуры, которая содержит только типы значений, необходимо реализовать метод `Clone()` с использованием метода `MemberwiseClone()`. Однако если есть специальный тип, поддерживающий ссылочные типы, тогда для построения глубокой копии может потребоваться создать новый объект, который учитывает каждую переменную-член ссылочного типа.

Интерфейс IComparable

Интерфейс `System.IComparable` описывает поведение, которое позволяет сортировать объекты на основе указанного ключа. Вот его формальное определение:

```
// Данный интерфейс позволяет объекту указывать
// его отношение с другими подобными объектами.
public interface IComparable
{
  int CompareTo(object o);
}
```

На заметку! Обобщенная версия этого интерфейса (`IComparable<T>`) предлагает более безопасный в отношении типов способ обработки операций сравнения объектов. Обобщения исследуются в главе 10.

Создайте новый проект консольного приложения по имени `ComparableCar`, скопируйте классы `Car` и `Radio` из проекта `SimpleException`, рассмотренного в главе 7, и поменяйте пространство имен в каждом файле класса на `ComparableCar`. Обновите класс `Car`, добавив новое свойство для представления уникального идентификатора каждого автомобиля и модифицированный конструктор:

```
using System;
using System.Collections;
namespace ComparableCar
{
  public class Car
  {
    ...
    public int CarID {get; set;}
    public Car(string name, int currSp, int id)
    {
      CurrentSpeed = currSp;
      PetName = name;
      CarID = id;
    }
    ...
  }
}
```

Теперь предположим, что имеется следующий массив объектов `Car`:

```
using System;
using ComparableCar;
Console.WriteLine("***** Fun with Object Sorting *****\n");

// Создать массив объектов Car.
Car[] myAutos = new Car[5];
myAutos[0] = new Car("Rusty", 80, 1);
myAutos[1] = new Car("Mary", 40, 234);
myAutos[2] = new Car("Viper", 40, 34);
myAutos[3] = new Car("Mel", 40, 4);
myAutos[4] = new Car("Chucky", 40, 5);

Console.ReadLine();
```

В классе `System.Array` определен статический метод по имени `Sort()`. Его вызов для массива внутренних типов (`int`, `short`, `string` и т.д.) приводит к сортировке элементов массива в числовом или алфавитном порядке, т.к. внутренние типы данных реализуют интерфейс `IComparable`. Но что произойдет, если передать методу `Sort()` массив объектов `Car`?

```
// Сортируются ли объекты Car? Пока еще нет!
Array.Sort(myAutos);
```

Запустив тестовый код, вы получите исключение времени выполнения, потому что класс `Car` не поддерживает необходимый интерфейс. При построении специальных типов вы можете реализовать интерфейс `IComparable`, чтобы позволить массивам, содержащим элементы этих типов, подвергаться сортировке. Когда вы реализуете де-

тали CompareTo(), то должны самостоятельно принять решение о том, что должно браться за основу в операции упорядочивания. Для типа Car вполне логичным кандидатом может служить внутреннее свойство CarID:

```
// Итерация по объектам Car может быть упорядочена на основе CarID.
public class Car : IComparable
{
  ...
  // Реализация интерфейса IComparable.
  int IComparable.CompareTo(object obj)
  {
    if (obj is Car temp)
    {
      if (this.CarID > temp.CarID)
      {
        return 1;
      }
      if (this.CarID < temp.CarID)
      {
        return -1;
      }
      return 0;
    }
    throw new ArgumentException("Parameter is not a Car!");
                              // Параметр не является объектом типа Car!
  }
}
```

Как видите, логика метода CompareTo() заключается в сравнении входного объекта с текущим экземпляром на основе специфичного элемента данных. Возвращаемое значение метода CompareTo() применяется для выяснения того, является текущий объект меньше, больше или равным объекту, с которым он сравнивается (табл. 8.1).

Таблица 8.1. Возвращаемые значения метода CompareTo()

Возвращаемое значение	Описание
Любое число меньше нуля	Этот экземпляр находится перед указанным объектом в порядке сортировки
Ноль	Этот экземпляр равен указанному объекту
Любое число больше нуля	Этот экземпляр находится после указанного объекта в порядке сортировки

Предыдущую реализацию метода CompareTo() можно усовершенствовать с учетом того факта, что тип данных int в C# (который представляет собой просто сокращенное обозначение для типа System.Int32) реализует интерфейс IComparable. Реализовать CompareTo() в Car можно было бы так:

```
int IComparable.CompareTo(object obj)
{
  if (obj is Car temp)
  {
    return this.CarID.CompareTo(temp.CarID);
  }
```

```
    throw new ArgumentException("Parameter is not a Car!");
                                // Параметр не является объектом типа Car!
}
```

В любом случае, поскольку тип `Car` понимает, как сравнивать себя с подобными объектами, вы можете написать следующий тестовый код:

```
// Использование интерфейса IComparable.
// Создать массив объектов Car.
...
// Отобразить текущее содержимое массива.
Console.WriteLine("Here is the unordered set of cars:");
foreach(Car c in myAutos)
{
  Console.WriteLine("{0} {1}", c.CarID, c.PetName);
}
// Теперь отсортировать массив с применением IComparable!
Array.Sort(myAutos);
Console.WriteLine();

// Отобразить отсортированное содержимое массива.
Console.WriteLine(«Here is the ordered set of cars:»);
foreach(Car c in myAutos)
{
  Console.WriteLine(«{0} {1}», c.CarID, c.PetName);
}
Console.ReadLine();
```

Ниже показан вывод, полученный в результате выполнения приведенного выше кода:

```
***** Fun with Object Sorting *****

Here is the unordered set of cars:
1 Rusty
234 Mary
34 Viper
4 Mel
5 Chucky

Here is the ordered set of cars:
1 Rusty
4 Mel
5 Chucky
34 Viper
234 Mary
```

Указание множества порядков сортировки с помощью `IComparer`

В текущей версии класса `Car` в качестве основы для порядка сортировки используется идентификатор автомобиля (`CarID`). В другом проектном решении основой сортировки могло быть дружественное имя автомобиля (для вывода списка автомобилей в алфавитном порядке). А что если вы хотите построить класс `Car`, который можно было бы подвергать сортировке по идентификатору *и также* по дружественному имени? В таком случае вы должны ознакомиться с еще одним стандартным интерфейсом по имени `IComparer`, который определен в пространстве имен `System.Collections` следующим образом:

```
// Общий способ сравнения двух объектов.
interface IComparer
{
  int Compare(object o1, object o2);
}
```

На заметку! Обобщенная версия этого интерфейса (`IComparer<T>`) обеспечивает более безопасный в отношении типов способ обработки операций сравнения объектов. Обобщения подробно рассматриваются в главе 10.

В отличие от `IComparable` интерфейс `IComparer` обычно *не* реализуется в типе, который вы пытаетесь сортировать (т.е. `Car`). Взамен данный интерфейс реализуется в любом количестве вспомогательных классов, по одному для каждого порядка сортировки (на основе дружественного имени, идентификатора автомобиля и т.д.). В настоящий момент типу `Car` уже известно, как сравнивать автомобили друг с другом по внутреннему идентификатору. Следовательно, чтобы позволить пользователю объекта сортировать массив объектов `Car` по дружественному имени, потребуется создать дополнительный вспомогательный класс, реализующий интерфейс `IComparer`. Вот необходимый код (не забудьте импортировать в файл кода пространство имен `System.Collections`):

```
using System;
using System.Collections;

namespace ComparableCar
{
  // Этот вспомогательный класс используется для сортировки
  // массива объектов Car по дружественному имени.
  public class PetNameComparer : IComparer
  {
    // Проверить дружественное имя каждого объекта.
    int IComparer.Compare(object o1, object o2)
    {
      if (o1 is Car t1 && o2 is Car t2)
      {
        return string.Compare(t1.PetName, t2.PetName,
          StringComparison.OrdinalIgnoreCase);
      }
      else
      {
        throw new ArgumentException("Parameter is not a Car!");
                                // Параметр не является объектом типа Car!
      }
    }
  }
}
```

Вспомогательный класс `PetNameComparer` может быть задействован в коде. Класс `System.Array` содержит несколько перегруженных версий метода `Sort()`, одна из которых принимает объект, реализующий интерфейс `IComparer`:

```
...
// Теперь сортировать по дружественному имени.
Array.Sort(myAutos, new PetNameComparer());
```

```
// Вывести отсортированный массив.
Console.WriteLine("Ordering by pet name:");
foreach(Car c in myAutos)
{
  Console.WriteLine("{0} {1}", c.CarID, c.PetName);
}
...
```

Специальные свойства и специальные типы сортировки

Важно отметить, что вы можете применять специальное статическое свойство, оказывая пользователю объекта помощь с сортировкой типов `Car` по специфичному элементу данных. Предположим, что в класс `Car` добавлено статическое свойство только для чтения по имени `SortByPetName`, которое возвращает экземпляр класса, реализующего интерфейс `IComparer` (в этом случае `PetNameComparer`; не забудьте импортировать пространство имен `System.Collections`):

```
// Теперь мы поддерживаем специальное свойство для возвращения
// корректного экземпляра, реализующего интерфейс IComparer.
public class Car : IComparable
{
  ...
  // Свойство, возвращающее PetNameComparer.
  public static IComparer SortByPetName
    => (IComparer)new PetNameComparer();
}
```

Теперь в коде массив можно сортировать по дружественному имени, используя жестко ассоциированное свойство, а не автономный класс `PetNameComparer`:

```
// Сортировка по дружественному имени становится немного яснее.
Array.Sort(myAutos, Car.SortByPetName);
```

К настоящему моменту вы должны не только понимать способы определения и реализации собственных интерфейсов, но также оценить их полезность. Конечно, интерфейсы встречаются внутри каждого важного пространства имен .NET Core, а в оставшихся главах книги вы продолжите работать с разнообразными стандартными интерфейсами.

Резюме

Интерфейс может быть определен как именованная коллекция *абстрактных членов*. Интерфейс общепринято расценивать как поведение, которое может поддерживаться заданным типом. Когда два или больше число типов реализуют один и тот же интерфейс, каждый из них может трактоваться одинаковым образом (полиморфизм на основе интерфейсов), даже если типы определены в разных иерархиях.

Для определения новых интерфейсов в языке C# предусмотрено ключевое слово `interface`. Как было показано в главе, тип может поддерживать столько интерфейсов, сколько необходимо, и интерфейсы указываются в виде списка с разделителями-запятыми. Более того, разрешено создавать интерфейсы, которые являются производными от множества базовых интерфейсов.

В дополнение к построению специальных интерфейсов библиотеки .NET Core определяют набор стандартных (т.е. поставляемых вместе с платформой) интерфейсов. Вы видели, что можно создавать специальные типы, которые реализуют предопределенные интерфейсы с целью поддержки набора желательных возможностей, таких как клонирование, сортировка и перечисление.

ГЛАВА 9

Время существования объектов

К настоящему моменту вы уже умеете создавать специальные типы классов в C#. Теперь вы узнаете, каким образом исполняющая среда управляет размещенными экземплярами классов (т.е. объектами) посредством *сборки мусора*. Программистам на C# никогда не приходится непосредственно удалять управляемый объект из памяти (вспомните, что в языке C# даже нет ключевого слова наподобие `delete`). Взамен объекты .NET Core размещаются в области памяти, которая называется *управляемой кучей*, где они автоматически уничтожаются сборщиком мусора "в какой-то момент в будущем".

После изложения основных деталей, касающихся процесса сборки мусора, будет показано, каким образом программно взаимодействовать со сборщиком мусора, используя класс `System.GC` (что в большинстве проектов обычно не требуется). Мы рассмотрим, как с применением виртуального метода `System.Object.Finalize()` и интерфейса `IDisposable` строить классы, которые своевременно и предсказуемо освобождают внутренние *неуправляемые ресурсы*.

Кроме того, будут описаны некоторые функциональные возможности сборщика мусора, появившиеся в версии .NET 4.0, включая фоновую сборку мусора и ленивое (отложенное) создание объектов с использованием обобщенного класса `System.Lazy<>`. После освоения материалов данной главы вы должны хорошо понимать, каким образом исполняющая среда управляет объектами .NET Core.

Классы, объекты и ссылки

Прежде чем приступить к исследованию основных тем главы, важно дополнительно прояснить отличие между классами, объектами и ссылочными переменными. Вспомните, что класс — всего лишь модель, которая описывает то, как экземпляр такого типа будет выглядеть и вести себя в памяти. Разумеется, классы определяются внутри файлов кода (которым по соглашению назначается расширение `*.cs`). Взгляните на следующий простой класс `Car`, определенный в новом проекте консольного приложения C# по имени `SimpleGC`:

```
// Car.cs
namespace SimpleGC
{
  public class Car
  {
```

```
    public int CurrentSpeed {get; set;}
    public string PetName {get; set;}

    public Car(){}
    public Car(string name, int speed)
    {
      PetName = name;
      CurrentSpeed = speed;
    }
    public override string ToString()
      => $"{PetName} is going {CurrentSpeed} MPH";
  }
}
```

После того как класс определен, в памяти можно размещать любое количество его объектов, применяя ключевое слово `new` языка C#. Однако следует иметь в виду, что ключевое слово `new` возвращает *ссылку* на объект в куче, а не действительный объект. Если ссылочная переменная объявляется как локальная переменная в области действия метода, то она сохраняется в стеке для дальнейшего использования внутри приложения. Для доступа к членам объекта в отношении сохраненной ссылки необходимо применять операцию точки C#:

```
using System;
using SimpleGC;
Console.WriteLine("***** GC Basics *****");

// Создать новый объект Car в управляемой куче.
// Возвращается ссылка на этот объект (refToMyCar).
Car refToMyCar = new Car("Zippy", 50);

// Операция точки (.) используется для обращения к членам
// объекта с применением ссылочной переменной.
Console.WriteLine(refToMyCar.ToString());
Console.ReadLine();
```

На рис. 9.1 показаны отношения между классами, объектами и ссылками.

На заметку! Вспомните из главы 4, что структуры являются типами значений, которые всегда размещаются прямо в стеке и никогда не попадают в управляемую кучу .NET Core. Размещение в куче происходит только при создании экземпляров классов.

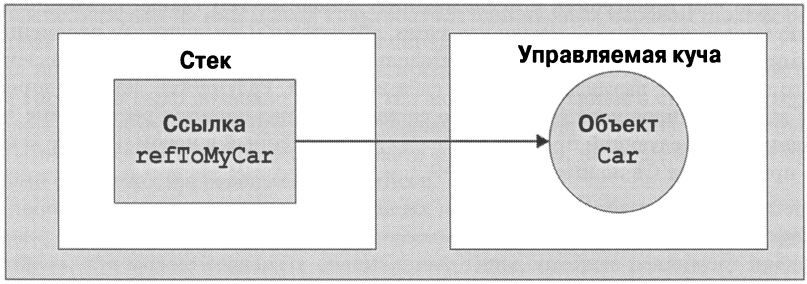

Рис. 9.1. Ссылки на объекты в управляемой куче

Базовые сведения о времени жизни объектов

При создании приложений C# корректно допускать, что исполняющая среда .NET Core позаботится об управляемой куче без вашего прямого вмешательства. В действительности "золотое правило" по управлению памятью в .NET Core выглядит простым.

Правило. Используя ключевое слово new, поместите экземпляр класса в управляемую кучу и забудьте о нем.

После создания объект будет автоматически удален сборщиком мусора, когда необходимость в нем отпадет. Конечно, возникает вполне закономерный вопрос о том, каким образом сборщик мусора выясняет, что объект больше не нужен? Краткий (т.е. неполный) ответ можно сформулировать так: сборщик мусора удаляет объект из кучи, только когда он становится *недостижимым* для любой части кодовой базы. Добавьте в класс `Program` метод, который размещает в памяти локальный объект `Car`:

```
static void MakeACar()
{
  // Если myCar - единственная ссылка на объект Car, то после
  // завершения этого метода объект Car *может* быть уничтожен.
  Car myCar = new Car();
}
```

Обратите внимание, что ссылка на объект `Car` (`myCar`) была создана непосредственно внутри метода `MakeACar()` и не передавалась за пределы определяющей области видимости (через возвращаемое значение или параметр `ref/out`). Таким образом, после завершения данного метода ссылка `myCar` оказывается недостижимой, и объект `Car` теперь является кандидатом на удаление сборщиком мусора. Тем не менее, важно понимать, что восстановление занимаемой этим объектом памяти немедленно после завершения метода `MakeACar()` гарантировать нельзя. В данный момент можно предполагать лишь то, что когда исполняющая среда инициирует следующую сборку мусора, объект `myCar` может быть безопасно уничтожен.

Как вы наверняка сочтете, программирование в среде со сборкой мусора значительно облегчает разработку приложений. И напротив, программистам на языке C++ хорошо известно, что если они не позаботятся о ручном удалении размещенных в куче объектов, тогда утечки памяти не заставят себя долго ждать. На самом деле отслеживание утечек памяти — один из требующих самых больших затрат времени (и утомительных) аспектов программирования в неуправляемых средах. За счет того, что сборщику мусора разрешено взять на себя заботу об уничтожении объектов, обязанности по управлению памятью перекладываются с программистов на исполняющую среду.

Код CIL для ключевого слова new

Когда компилятор C# сталкивается с ключевым словом `new`, он вставляет в реализацию метода инструкцию `newobj` языка CIL. Если вы скомпилируете текущий пример кода и заглянете в полученную сборку с помощью утилиты `ildasm.exe`, то найдете внутри метода `MakeACar()` следующие операторы CIL:

```
.method assembly hidebysig static
       void '<<Main>$>g__MakeACar|0_0'() cil managed
{
```

```
  // Code size        8 (0x8)
  // Размер кода      8 (0x8)
  .maxstack 1
  .locals init (class SimpleGC.Car V_0)
  IL_0000: nop
  IL_0001: newobj instance void SimpleGC.Car::.ctor()
  IL_0006: stloc.0
  IL_0007: ret
} // end of method '<Program>$'::'<<Main>$>g__MakeACar|0_0'
  // конец метода '<Program>$'::'<<Main>$>g__MakeACar|0_0'
```

Прежде чем ознакомиться с точными правилами, которые определяют момент, когда объект должен удаляться из управляемой кучи, давайте более подробно рассмотрим роль инструкции `newobj` языка CIL. Первым делом важно понимать, что управляемая куча представляет собой нечто большее, чем просто произвольную область памяти, к которой исполняющая среда имеет доступ. Сборщик мусора .NET Core "убирает" кучу довольно тщательно, при необходимости даже сжимая пустые блоки памяти в целях оптимизации.

Для содействия его усилиям в управляемой куче поддерживается указатель (обычно называемый *указателем на следующий объект* или *указателем на новый объект*), который идентифицирует точное местоположение, куда будет помещен следующий объект. Таким образом, инструкция `newobj` заставляет исполняющую среду выполнить перечисленные ниже основные операции.

1. Подсчитать общий объем памяти, требуемой для размещения объекта (в том числе память, необходимую для членов данных и базовых классов).
2. Выяснить, действительно ли в управляемой куче имеется достаточно пространства для сохранения размещаемого объекта. Если места хватает, то указанный конструктор вызывается, и вызывающий код в конечном итоге получает ссылку на новый объект в памяти, адрес которого совпадает с последней позицией указателя на следующий объект.
3. Наконец, перед возвращением ссылки вызывающему коду переместить указатель на следующий объект, чтобы он указывал на следующую доступную область в управляемой куче.

Описанный процесс проиллюстрирован на рис. 9.2.

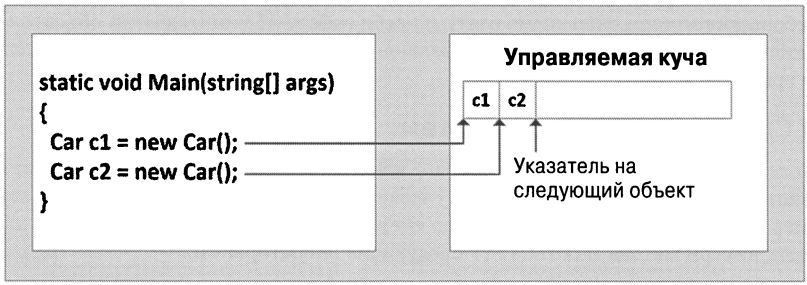

Рис. 9.2. Детали размещения объектов в управляемой куче

В результате интенсивного размещения объектов приложением пространство внутри управляемой кучи может со временем заполниться. Если при обработке инструкции newobj исполняющая среда определяет, что в управляемой куче недостаточно места для размещения объекта запрашиваемого типа, тогда она выполнит сборку мусора, пытаясь освободить память. Соответственно, следующее правило сборки мусора выглядит тоже довольно простым.

Правило. Если в управляемой куче не хватает пространства для размещения требуемого объекта, то произойдет сборка мусора.

Однако то, *как* конкретно происходит сборка мусора, зависит от типа сборки мусора, используемого приложением. Различия будут описаны далее в главе.

Установка объектных ссылок в null

Программисты на C/C++ часто устанавливают переменные указателей в null, гарантируя тем самым, что они больше не ссылаются на какие-то местоположения в неуправляемой памяти. Учитывая такой факт, вас может интересовать, что происходит в результате установки в null ссылок на объекты в C#. В качестве примера измените метод MakeACar() следующим образом:

```
static void MakeACar()
{
  Car myCar = new Car();
  myCar = null;
}
```

Когда ссылке на объект присваивается null, компилятор C# генерирует код CIL, который гарантирует, что ссылка (myCar в данном примере) больше не указывает на какой-либо объект. Если теперь снова с помощью утилиты ildasm.exe просмотреть код CIL модифицированного метода MakeACar(), то можно обнаружить в нем код операции ldnull (заталкивает значение null в виртуальный стек выполнения), за которым следует код операции stloc.0 (устанавливает для переменной ссылку null):

```
.method assembly hidebysig static
       void '<<Main>$>g__MakeACar|0_0'() cil managed
{
  // Code size       10 (0xa)
  .maxstack 1
  .locals init (class SimpleGC.Car V_0)
  IL_0000: nop
  IL_0001: newobj instance void SimpleGC.Car::.ctor()
  IL_0006: stloc.0
  IL_0007: ldnull
  IL_0008: stloc.0
  IL_0009: ret
} // end of method '<Program>$'::'<<Main>$>g__MakeACar|0_0'
```

Тем не менее, вы должны понимать, что присваивание ссылке значения null ни в коей мере не вынуждает сборщик мусора немедленно запуститься и удалить объект из кучи. Единственное, что при этом достигается — явный разрыв связи между ссылкой и объектом, на который она ранее указывала. Таким образом, установка ссылок в null в C# имеет гораздо меньше последствий, чем в других языках, основанных на C; однако никакого вреда она определенно не причиняет.

Выяснение, нужен ли объект

Теперь вернемся к вопросу о том, как сборщик мусора определяет момент, когда объект больше не нужен. Для выяснения, активен ли объект, сборщик мусора использует следующую информацию.

- Корневые элементы в стеке: переменные в стеке, предоставляемые компилятором и средством прохода по стеку.
- Дескрипторы сборки мусора: дескрипторы, указывающие на объекты, на которые можно ссылаться из кода или исполняющей среды.
- Статические данные: статические объекты в доменах приложений, которые могут ссылаться на другие объекты.

Во время процесса сборки мусора исполняющая среда будет исследовать объекты в управляемой куче с целью выяснения, являются ли они по-прежнему достижимыми (т.е. корневыми) для приложения. Для такой цели исполняющая среда будет строить *граф объектов*, который представляет каждый достижимый объект в куче. Более подробно графы объектов объясняются во время рассмотрения сериализации объектов в главе 20. Пока достаточно знать, что графы объектов применяются для документирования всех достижимых объектов. Кроме того, имейте в виду, что сборщик мусора никогда не будет создавать граф для того же самого объекта дважды, избегая необходимости в выполнении утомительного подсчета циклических ссылок, который характерен при программировании COM.

Предположим, что в управляемой куче находится набор объектов с именами A, B, C, D, E, F и G. Во время сборки мусора эти объекты (а также любые внутренние объектные ссылки, которые они могут содержать) будут проверяться. После построения графа недостижимые объекты (пусть ими будут объекты C и F) помечаются как являющиеся мусором. На рис. 9.3 показан возможный граф объектов для только что описанного сценария (линии со стрелками можно читать как "зависит от" или "требует", т.е. E зависит от G и B, A не зависит ни от чего и т.д.).

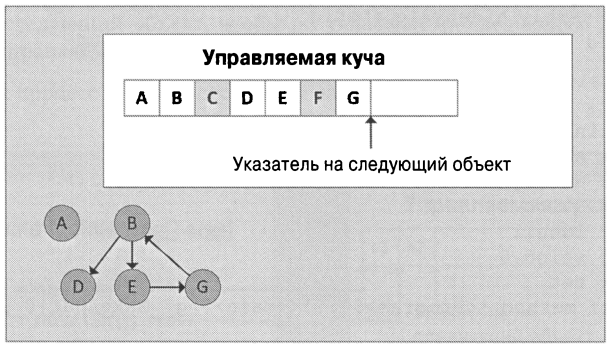

Рис. 9.3. Графы объектов строятся с целью определения объектов, достижимых для корневых элементов приложения

После того как объекты помечены для уничтожения (в данном случае C и F, т.к. они не учтены в графе объектов), они удаляются из памяти. Оставшееся пространст-

во в куче сжимается, что в свою очередь вынуждает исполняющую среду изменить лежащие в основе указатели для ссылки на корректные местоположения в памяти (это делается автоматически и прозрачно). И последнее, но не менее важное действие — указатель на следующий объект перенастраивается так, чтобы указывать на следующую доступную область памяти. Конечный результат описанных изменений представлен на рис. 9.4.

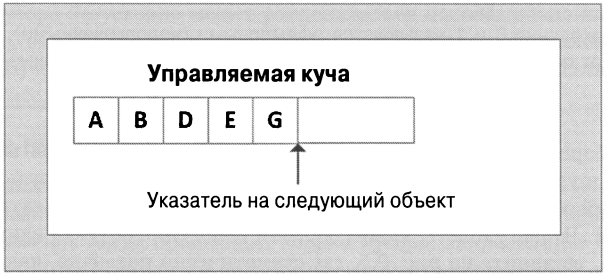

Рис. 9.4. Очищенная и сжатая куча

На заметку! Строго говоря, сборщик мусора использует две отдельные кучи, одна из которых предназначена специально для хранения крупных объектов. Во время сборки мусора обращение к данной куче производится менее часто из-за возможного снижения производительности, связанного с перемещением больших объектов. В .NET Core куча для хранения крупных объектов может быть уплотнена по запросу или при достижении необязательных жестких границ, устанавливающих абсолютную или процентную степень использования памяти.

Понятие поколений объектов

Когда исполняющая среда пытается найти недостижимые объекты, она не проверяет буквально каждый объект, помещенный в управляемую кучу. Очевидно, это потребовало бы значительного времени, тем более в крупных (т.е. реальных) приложениях.

Для содействия оптимизации процесса каждому объекту в куче назначается специфичное "поколение". Лежащая в основе поколений идея проста: чем дольше объект существует в куче, тем выше вероятность того, что он там будет оставаться. Например, класс, который определяет главное окно настольного приложения, будет находиться в памяти вплоть до завершения приложения. С другой стороны объекты, которые были помещены в кучу только недавно (такие как объект, размещенный внутри области действия метода), по всей видимости, довольно быстро станут недостижимыми. Исходя из таких предположений, каждый объект в куче принадлежит совокупности одного из перечисленных ниже поколений.

- *Поколение 0.* Идентифицирует новый размещенный в памяти объект, который еще никогда не помечался как подлежащий сборке мусора (за исключением крупных объектов, изначально помещаемых в совокупность поколения 2). Большинство объектов утилизируются сборщиком мусора в поколении 0 и не доживают до поколения 1.

- *Поколение 1.* Идентифицирует объект, который уже пережил одну сборку мусора. Это поколение также служит буфером между кратко и длительно существующими объектами.
- *Поколение 2.* Идентифицирует объект, которому удалось пережить более одной очистки сборщиком мусора, или весьма крупный объект, появившийся в совокупности поколения 2.

На заметку! Поколения 0 и 1 называются эфемерными (недолговечными). В следующем разделе будет показано, что процесс сборки мусора трактует эфемерные поколения по-разному.

Сначала сборщик мусора исследует все объекты, относящиеся к поколению 0. Если пометка и удаление (или освобождение) таких объектов в результате обеспечивают требуемый объем свободной памяти, то любые уцелевшие объекты повышаются до поколения 1. Чтобы увидеть, каким образом поколение объекта влияет на процесс сборки мусора, взгляните на рис. 9.5, где схематически показано, как набор уцелевших объектов поколения 0 (A, B и E) повышается до следующего поколения после восстановления требуемого объема памяти.

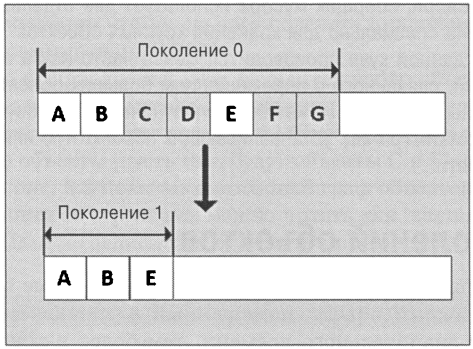

Рис. 9.5. Объекты поколения 0, которые уцелели после сборки мусора, повышаются до поколения 1

Если все объекты поколения 0 проверены, но по-прежнему требуется дополнительная память, тогда начинают исследоваться на предмет достижимости и подвергаться сборке мусора объекты поколения 1. Уцелевшие объекты поколения 1 повышаются до поколения 2. Если же сборщику мусора *все еще* требуется дополнительная память, то начинают проверяться объекты поколения 2. На этом этапе объекты поколения 2, которым удается пережить сборку мусора, остаются объектами того же поколения 2, учитывая заранее определенный верхний предел поколений объектов.

В заключение следует отметить, что за счет назначения объектам в куче определенного поколения более новые объекты (такие как локальные переменные) будут удаляться быстрее, в то время как более старые (наподобие главного окна приложения) будут существовать дольше.

Сборка мусора инициируется, когда в системе оказывается мало физической памяти, когда объем памяти, выделенной в физической куче, превышает приемлемый порог или когда в коде приложения вызывается метод `GC.Collect()`.

Если все описанное выглядит слегка удивительным и более совершенным, чем необходимость в самостоятельном управлении памятью, тогда имейте в виду, что процесс сборки мусора не обходится без определенных затрат. Время сборки мусора и то, что будет подвергаться сборке, обычно не находится под контролем разработчиков, хотя сборка мусора безусловно может расцениваться положительно или отрицательно. Кроме того, выполнение сборки мусора приводит к расходу циклов центрального процессора (ЦП) и может повлиять на производительность приложения. В последующих разделах исследуются различные типы сборки мусора.

Эфемерные поколения и сегменты

Как упоминалось ранее, поколения 0 и 1 существуют недолго и называются *эфемерными поколениями*. Эти поколения размещаются в памяти, которая известна как *эфемерный сегмент*. Когда происходит сборка мусора, запрошенные сборщиком мусора новые сегменты становятся новыми эфемерными сегментами, а сегменты, содержащие объекты, которые уцелели в прошедшем поколении 1, образуют новый сегмент поколения 2. Размер эфемерного сегмента зависит от ряда факторов, таких как тип сборки мусора (рассматривается следующим) и разрядность системы. Размеры эфемерных сегментов описаны в табл. 9.1.

Таблица 9.1 Размеры эфемерных сегментов

Тип сборки мусора	32 разряда	64 разряда
Сборка мусора на рабочей станции	16 Мбайт	256 Мбайт
Сборка мусора на сервере	64 Мбайт	4 Гбайт
Сборка мусора на сервере с числом логических ЦП больше 4	32 Мбайт	2 Гбайт
Сборка мусора на сервере с числом логических ЦП больше 8	16 Мбайт	1 Гбайт

Типы сборки мусора

Исполняющая среда поддерживает два описанных ниже типа сборки мусора.

- *Сборка мусора на рабочей станции.* Тип сборки мусора, который спроектирован для клиентских приложений и является стандартным для автономных приложений. Сборка мусора на рабочей станции может быть фоновой (обсуждается ниже) или выполняться в непараллельном режиме.
- *Сборка мусора на сервере.* Тип сборки мусора, спроектированный для серверных приложений, которым требуется высокая производительность и масштабируемость. Подобно сборке мусора на рабочей станции сборка мусора на сервере может быть фоновой или выполняться в непараллельном режиме.

На заметку! Названия служат признаком стандартных настроек для приложений рабочей станции и сервера, но метод сборки мусора можно настраивать через файл `runtimeconfig.json` или переменные среды системы. При наличии на компьютере только одного ЦП будет всегда использоваться сборка мусора на рабочей станции.

Сборка мусора на рабочей станции производится в том же потоке, где она была инициирована, и сохраняет тот же самый приоритет, который был назначен во время запуска. Это может привести к состязанию с другими потоками в приложении.

Сборка мусора на сервере осуществляется в нескольких выделенных потоках, которым назначен уровень приоритета `THREAD_PRIORITY_HIGHEST` (тема многопоточности раскрывается в главе 15). Для выполнения сборки мусора каждый ЦП получает выделенную кучу и отдельный поток. В итоге сборка мусора на сервере может стать крайне ресурсоемкой.

Фоновая сборка мусора

Начиная с версии .NET 4.0 и продолжая в .NET Core, сборщик мусора способен решать вопрос с приостановкой потоков при очистке объектов в управляемой куче, используя *фоновую сборку мусора*. Несмотря на название приема, это вовсе не означает, что вся сборка мусора теперь происходит в дополнительных фоновых потоках выполнения. На самом деле, если фоновая сборка мусора производится для объектов, принадлежащих к неэфемерному поколению, то исполняющая среда .NET Core может выполнять сборку мусора в отношении объектов эфемерных поколений внутри отдельного фонового потока.

В качестве связанного замечания: механизм сборки мусора в .NET 4.0 и последующих версиях был усовершенствован с целью дальнейшего сокращения времени приостановки заданного потока, которая связана со сборкой мусора. Конечным результатом таких изменений стало то, что процесс очистки неиспользуемых объектов поколения 0 или поколения 1 был оптимизирован и позволяет обеспечить более высокую производительность приложений (что действительно важно для систем реального времени, которые требуют небольших и предсказуемых перерывов на сборку мусора).

Тем не менее, важно понимать, что ввод новой модели сборки мусора совершенно не повлиял на способ построения приложений .NET Core. С практической точки зрения вы можете просто разрешить сборщику мусора выполнять свою работу без непосредственного вмешательства с вашей стороны (и радоваться тому, что разработчики в Microsoft продолжают улучшать процесс сборки мусора в прозрачной манере).

Тип `System.GC`

В сборке `mscorlib.dll` предоставляется класс по имени `System.GC`, который позволяет программно взаимодействовать со сборщиком мусора, применяя набор статических членов. Имейте в виду, что необходимость в прямом взаимодействии с классом `System.GC` внутри разрабатываемого кода возникает редко (если вообще возникает). Обычно единственной ситуацией, когда будут использоваться члены `System.GC`, является создание классов, которые внутренне работают с *неуправляемыми ресурсами*. Например, может строиться класс, в котором присутствуют вызовы API-интерфейса Windows, основанного на C, с применением протокола обращения к платформе .NET Core, или какая-то низкоуровневая и сложная логика взаимодействия с COM. В табл. 9.2 приведено краткое описание некоторых наиболее интересных членов класса `System.GC` (полные сведения можно найти в документации по .NET Core).

Таблица 9.2. Избранные члены типа `System.GC`

Члены `System.GC`	Описание
`AddMemoryPressure()` `RemoveMemoryPressure()`	Позволяют указывать числовое значение, которое представляет "уровень срочности" (или давление) вызывающего объекта относительно процесса сборки мусора. Имейте в виду, что эти методы должны изменять уровень давления *в тандеме*; следовательно, нельзя удалять более высокий показатель давления, чем тот, который был добавлен
`Collect()`	Заставляет сборщик мусора выполнить сборку мусора. Этот метод перегружен для указания поколения, подлежащего сборке, а также режима сборки (посредством перечисления `GCCollectionMode`)
`CollectionCount()`	Возвращает числовое значение, которое показывает, сколько раз производилась сборка мусора для заданного поколения
`GetGeneration()`	Возвращает поколение, к которому относится объект в текущий момент
`GetTotalMemory()`	Возвращает оценочный объем памяти (в байтах), выделенной в управляемой куче в текущий момент. Булевский параметр указывает, должен ли вызов дождаться выполнения сборки мусора перед возвращением
`MaxGeneration`	Возвращает максимальное количество поколений, поддерживаемое целевой системой. Начиная с версии .NET 4.0, есть три возможных поколения: 0, 1 и 2
`SuppressFinalize()`	Устанавливает флаг, который указывает, что заданный объект не должен вызывать свой метод `Finalize()`
`WaitForPendingFinalizers()`	Приостанавливает текущий поток до тех пор, пока не будут финализированы все финализируемые объекты. Обычно вызывается сразу после вызова метода `GC.Collect()`

Чтобы проиллюстрировать использование типа `System.GC` для получения разнообразных деталей, связанных со сборкой мусора, обновите операторы верхнего уровня в проекте `SimpleGC`:

```
using System;
Console.WriteLine("***** Fun with System.GC *****");

// Вывести оценочное количество байтов, выделенных в куче.
Console.WriteLine("Estimated bytes on heap: {0}",
  GC.GetTotalMemory(false));

// Значения MaxGeneration начинаются с 0, поэтому при выводе добавить 1.
Console.WriteLine("This OS has {0} object generations.\n",
  (GC.MaxGeneration + 1));
Car refToMyCar = new Car("Zippy", 100);
Console.WriteLine(refToMyCar.ToString());

// Вывести поколение объекта refToMyCar.
Console.WriteLine("Generation of refToMyCar is: {0}",
  GC.GetGeneration(refToMyCar));
Console.ReadLine();
```

Вы должны получить примерно такой вывод:

```
***** Fun with System.GC *****

Estimated bytes on heap: 75760
This OS has 3 object generations.

Zippy is going 100 MPH
Generation of refToMyCar is: 0
```

Методы из табл. 9.2 более подробно обсуждаются в следующем разделе.

Принудительный запуск сборщика мусора

Не забывайте о том, что основное предназначение сборщика мусора связано с управлением памятью вместо программистов. Однако в ряде редких обстоятельств сборщик мусора полезно запускать принудительно, используя метод `GC.Collect()`. Взаимодействие с процессом сборки мусора требуется в двух ситуациях:

- приложение входит в блок кода, который не должен быть прерван вероятной сборкой мусора;
- приложение только что закончило размещение исключительно большого количества объектов, и вы хотите насколько возможно скоро освободить крупный объем выделенной памяти.

Если вы посчитаете, что принудительная проверка сборщиком мусора наличия недостижимых объектов может принести пользу, тогда можете явно инициировать процесс сборки мусора:

```
...
// Принудительно запустить сборку мусора
// и ожидать финализации каждого объекта.
GC.Collect();
GC.WaitForPendingFinalizers();
...
```

При запуске сборки мусора вручную всегда должен вызываться метод `GC.WaitForPendingFinalizers()`. Благодаря такому подходу можно иметь уверенность в том, что все *финализируемые объекты* (описанные в следующем разделе) получат шанс выполнить любую необходимую очистку перед продолжением работы программы. "За кулисами" метод `GC.WaitForPendingFinalizers()` приостановит вызывающий поток на время прохождения сборки мусора. Это очень хорошо, т.к. гарантирует невозможность обращения в коде к методам объекта, который в текущий момент уничтожается.

Методу `GC.Collect()` можно также предоставить числовое значение, идентифицирующее самое старое поколение, для которого будет выполняться сборка мусора. Например, чтобы проинструктировать исполняющую среду о необходимости исследования только объектов поколения 0, можно написать такой код:

```
...
// Исследовать только объекты поколения 0.
GC.Collect(0);
GC.WaitForPendingFinalizers();
...
```

Кроме того, методу `Collect()` можно передать во втором параметре значение перечисления `GCCollectionMode` для точной настройки способа, которым исполняющая среда должна принудительно инициировать сборку мусора. Ниже показаны значения, определенные этим перечислением:

```
public enum GCCollectionMode
{
  Default,      // Текущим стандартным значением является Forced.
  Forced,       // Указывает исполняющей среде начать сборку мусора
                // немедленно.
  Optimized     // Позволяет исполняющей среде выяснить, оптимален
                // ли текущий момент для удаления объектов.
}
```

Как и при любой сборке мусора, в результате вызова `GC.Collect()` уцелевшие объекты переводятся в более высокие поколения. Модифицируйте операторы верхнего уровня следующим образом:

```
Console.WriteLine("***** Fun with System.GC *****");

// Вывести оценочное количество байтов, выделенных в куче.
Console.WriteLine("Estimated bytes on heap: {0}",
 GC.GetTotalMemory(false));
// Значения MaxGeneration начинаются с 0.
Console.WriteLine("This OS has {0} object generations.\n",
  (GC.MaxGeneration + 1));
Car refToMyCar = new Car("Zippy", 100);
Console.WriteLine(refToMyCar.ToString());
// Вывести поколение refToMyCar.
Console.WriteLine("\nGeneration of refToMyCar is: {0}",
  GC.GetGeneration(refToMyCar));
// Создать большое количество объектов для целей тестирования.
object[] tonsOfObjects = new object[50000];
for (int i = 0; i < 50000; i++)
{
  tonsOfObjects[i] = new object();
}
// Выполнить сборку мусора только для объектов поколения 0.
Console.WriteLine("Force Garbage Collection");
GC.Collect(0, GCCollectionMode.Forced);
GC.WaitForPendingFinalizers();
// Вывести поколение refToMyCar.
Console.WriteLine("Generation of refToMyCar is: {0}",
GC.GetGeneration(refToMyCar));
// Посмотреть, существует ли еще tonsOfObjects[9000].
if (tonsOfObjects[9000] != null)
{
  Console.WriteLine("Generation of tonsOfObjects[9000] is: {0}",
    GC.GetGeneration(tonsOfObjects[9000]));
}
else
{
  Console.WriteLine("tonsOfObjects[9000] is no longer alive.");
                  // tonsOfObjects[9000] больше не существует
}
// Вывести количество проведенных сборок мусора для разных поколений.
  Console.WriteLine("\nGen 0 has been swept {0} times",
    GC.CollectionCount(0));  // Количество сборок для поколения 0
  Console.WriteLine("Gen 1 has been swept {0} times",
    GC.CollectionCount(1));  // Количество сборок для поколения 1
```

```
    Console.WriteLine("Gen 2 has been swept {0} times",
      GC.CollectionCount(2));  // Количество сборок для поколения 2
    Console.ReadLine();
  }
```

Здесь в целях тестирования преднамеренно был создан большой массив типа `object` (состоящий из 50 000 элементов). Ниже показан вывод программы:

```
***** Fun with System.GC *****

Estimated bytes on heap: 75760
This OS has 3 object generations.

Zippy is going 100 MPH
Generation of refToMyCar is: 0
Forcing Garbage Collection
Generation of refToMyCar is: 1
Generation of tonsOfObjects[9000] is: 1

Gen 0 has been swept 1 times
Gen 1 has been swept 0 times
Gen 2 has been swept 0 times
```

К настоящему моменту вы должны лучше понимать детали жизненного цикла объектов. В следующем разделе мы продолжим изучение процесса сборки мусора, обратившись к теме создания *финализируемых объектов* и *освобождаемых объектов*. Имейте в виду, что описываемые далее приемы обычно необходимы только при построении классов C#, которые поддерживают внутренние неуправляемые ресурсы.

Построение финализируемых объектов

В главе 6 вы узнали, что в самом главном базовом классе .NET Core, `System.Object`, определен виртуальный метод по имени `Finalize()`. В своей стандартной реализации он ничего не делает:

```
// System.Object
public class Object
{
  ...
  protected virtual void Finalize() {}
}
```

За счет переопределения метода `Finalize()` в специальных классах устанавливается специфическое место для выполнения любой логики очистки, необходимой данному типу. Учитывая, что метод `Finalize()` определен как защищенный, вызывать его напрямую из экземпляра класса через операцию точки нельзя. Взамен метод `Finalize()`, если он поддерживается, будет вызываться *сборщиком мусора* перед удалением объекта из памяти.

На заметку! Переопределять метод `Finalize()` в типах структур не разрешено. Подобное ограничение вполне логично, поскольку структуры являются типами значений, которые изначально никогда не размещаются в куче и, следовательно, никогда не подвергаются сборке мусора. Тем не менее, при создании структуры, которая содержит неуправляемые ресурсы, нуждающиеся в очистке, можно реализовать интерфейс `IDisposable` (вскоре он будет описан). Вспомните из главы 4, что структуры `ref` и структуры `ref`, допускающие только чтение, не могут реализовывать какой-либо интерфейс, но могут реализовывать метод `Dispose()`.

Разумеется, вызов метода `Finalize()` будет происходить (в итоге) во время "естественной" сборки мусора или в случае ее принудительного запуска внутри кода с помощью `GC.Collect()`. В предшествующих версиях .NET (но не в .NET Core) финализатор каждого объекта вызывался при окончании работы приложения. В .NET Core нет никаких способов принудительного запуска финализатора даже при завершении приложения.

О чем бы ни говорили ваши инстинкты разработчика, подавляющее большинство классов C# не требует написания явной логики очистки или специального финализатора. Причина проста: если в классах используются лишь другие управляемые объекты, то все они в конечном итоге будут подвергнуты сборке мусора. Единственная ситуация, когда может возникнуть потребность спроектировать класс, способный выполнять после себя очистку, предусматривает работу с *неуправляемыми* ресурсами (такими как низкоуровневые файловые дескрипторы операционной системы, низкоуровневые неуправляемые подключения к базам данных, фрагменты неуправляемой памяти и т.д.).

В рамках платформы .NET Core неуправляемые ресурсы получаются путем прямого обращения к API-интерфейсу операционной системы с применением служб вызова платформы (Platform Invocation Services — P/Invoke) или в сложных сценариях взаимодействия с COM. С учетом сказанного можно сформулировать еще одно правило сборки мусора.

Правило. Единственная серьезная причина для переопределения метода `Finalize()` связана с использованием в классе C# неуправляемых ресурсов через P/Invoke или сложные задачи взаимодействия с COM (обычно посредством разнообразных членов типа `System.Runtime.InteropServices.Marshal`). Это объясняется тем, что в таких сценариях производится манипулирование памятью, которой исполняющая среда управлять не может.

Переопределение метода `System.Object.Finalize()`

В том редком случае, когда строится класс C#, в котором применяются неуправляемые ресурсы, вы вполне очевидно захотите обеспечить предсказуемое освобождение занимаемой памяти. В качестве примера создадим новый проект консольного приложения C# по имени `SimpleFinalize` и вставим в него класс `MyResourceWrapper`, в котором используется неуправляемый ресурс (каким бы он ни был). Теперь необходимо переопределить метод `Finalize()`. Как ни странно, для этого нельзя применять ключевое слово `override` языка C#:

```
using System;
namespace SimpleFinalize
{
  class MyResourceWrapper
  {
    // Ошибка на этапе компиляции!
    protected override void Finalize() {}
  }
}
```

На самом деле для обеспечения того же самого эффекта используется синтаксис деструктора (подобный C++). Причина такой альтернативной формы переопределения виртуального метода заключается в том, что при обработке синтаксиса финализатора компилятор автоматически добавляет внутрь неявно переопределяемого метода `Finalize()` много обязательных инфраструктурных элементов (как вскоре будет показано).

Финализаторы C# выглядят похожими на конструкторы тем, что именуются идентично классу, в котором определены. Вдобавок они снабжаются префиксом в виде тильды (~). Однако в отличие от конструкторов финализаторы никогда не получают модификатор доступа (они всегда неявно защищенные), не принимают параметров и не могут быть перегружены (в каждом классе допускается наличие только одного финализатора). Ниже приведен специальный финализатор для класса `MyResourceWrapper`, который при вызове выдает звуковой сигнал. Очевидно, такой пример предназначен только для демонстрационных целей. В реальном приложении финализатор только освобождает любые неуправляемые ресурсы и *не* взаимодействует с другими управляемыми объектами, даже с теми, на которые ссылается текущий объект, т.к. нельзя предполагать, что они все еще существуют на момент вызова этого метода `Finalize()` сборщиком мусора.

```
using System;
// Переопределить System.Object.Finalize()
// посредством синтаксиса финализатора.
class MyResourceWrapper
{
  // Очистить неуправляемые ресурсы.
  // Выдать звуковой сигнал при уничтожении
  // (только в целях тестирования).
  ~MyResourceWrapper() => Console.Beep();
}
```

Если теперь просмотреть код CIL данного финализатора с помощью утилиты `ildasm.exe`, то обнаружится, что компилятор добавил необходимый код для проверки ошибок. Первым делом операторы внутри области действия метода `Finalize()` помещены в блок `try` (см. главу 7). Связанный с ним блок `finally` гарантирует, что методы `Finalize()` базовых классов будут всегда выполняться независимо от любых исключений, возникших в области `try`.

```
.method family hidebysig virtual instance void
Finalize() cil managed
{
  .override [System.Runtime]System.Object::Finalize
  // Code size      17 (0x11)
  .maxstack 1
  .try
  {
    IL_0000: call void [System.Console]System.Console::Beep()
    IL_0005: nop
    IL_0006: leave.s IL_0010
  } // end .try
  finally
  {
    IL_0008: ldarg.0
    IL_0009: call instance void [System.Runtime]System.Object::Finalize()
    IL_000e: nop
    IL_000f: endfinally
  } // end handler
  IL_0010: ret
} // end of method MyResourceWrapper::Finalize
```

Тестирование класса `MyResourceWrapper` показывает, что звуковой сигнал выдается при выполнении финализатора:

```
using System;
using SimpleFinalize;
Console.WriteLine("***** Fun with Finalizers *****\n");
Console.WriteLine("Hit return to create the objects ");
Console.WriteLine("then force the GC to invoke Finalize()");
          // Нажмите клавишу <Enter>, чтобы создать объекты
          // и затем заставить сборщик мусора вызвать метод Finalize()

// В зависимости от мощности вашей системы
// вам может понадобиться увеличить эти значения.
CreateObjects(1_000_000);

// Искусственно увеличить уровень давления.
GC.AddMemoryPressure(2147483647);
GC.Collect(0, GCCollectionMode.Forced);
GC.WaitForPendingFinalizers();
Console.ReadLine();

static void CreateObjects(int count)
{
  MyResourceWrapper[] tonsOfObjects = new MyResourceWrapper[count];
  for (int i = 0; i < count; i++)
  {
    tonsOfObjects[i] = new MyResourceWrapper();
  }
  tonsOfObjects = null;
}
```

На заметку! Единственный способ гарантировать, что такое небольшое консольное приложение принудительно запустит сборку мусора в .NET Core, предусматривает создание огромного количества объектов в памяти и затем установит ссылку на них в `null`. После запуска этого приложения не забудьте нажать комбинацию клавиш <Ctrl+C>, чтобы остановить его выполнение и прекратить выдачу звуковых сигналов!

Подробности процесса финализации

Важно всегда помнить о том, что роль метода `Finalize()` состоит в обеспечении того, что объект .NET Core сумеет освободить неуправляемые ресурсы, когда он подвергается сборке мусора. Таким образом, если вы строите класс, в котором неуправляемая память не применяется (общепризнанно самый распространенный случай), то финализация принесет мало пользы. На самом деле по возможности вы должны проектировать свои типы так, чтобы избегать в них поддержки метода `Finalize()` по той простой причине, что финализация занимает время.

При размещении объекта в управляемой куче исполняющая среда автоматически определяет, поддерживает ли он специальный метод `Finalize()`. Если да, тогда объект помечается как *финализируемый*, а указатель на него сохраняется во внутренней очереди, называемой *очередью финализации*. Очередь финализации — это таблица, обслуживаемая сборщиком мусора, в которой содержатся указатели на все объекты, подлежащие финализации перед удалением из кучи.

Когда сборщик мусора решает, что наступило время высвободить объект из памяти, он просматривает каждую запись в очереди финализации и копирует объект из кучи в еще одну управляемую структуру под названием *таблица объектов, доступных для финализации*. На этой стадии порождается отдельный поток для вызова метода `Finalize()` на каждом объекте из упомянутой таблицы *при следующей сборке мусора*. Итак, действительная финализация объекта требует, по меньшей мере, двух сборок мусора.

Подводя итоги, следует отметить, что хотя финализация объекта гарантирует ему возможность освобождения неуправляемых ресурсов, она все равно остается недетерминированной по своей природе, а из-за незаметной дополнительной обработки протекает значительно медленнее.

Построение освобождаемых объектов

Как вы уже видели, финализаторы могут использоваться для освобождения неуправляемых ресурсов при запуске сборщика мусора. Тем не менее, учитывая тот факт, что многие неуправляемые объекты являются "ценными элементами" (вроде низкоуровневых дескрипторов для файлов или подключений к базам данных), зачастую полезно их освобождать как можно раньше, не дожидаясь наступления сборки мусора. В качестве альтернативы переопределению метода `Finalize()` класс может реализовать интерфейс `IDisposable`, в котором определен единственный метод по имени `Dispose()`:

```
public interface IDisposable
{
  void Dispose();
}
```

При реализации интерфейса `IDisposable` предполагается, что когда *пользователь объекта* завершает с ним работу, он вручную вызывает метод `Dispose()` перед тем, как позволить объектной ссылке покинуть область действия. Таким способом объект может производить любую необходимую очистку неуправляемых ресурсов без помещения в очередь финализации и ожидания, пока сборщик мусора запустит логику финализации класса.

На заметку! Интерфейс `IDisposable` может быть реализован структурами не `ref` и классами (в отличие от переопределения метода `Finalize()`, что допускается только для классов), т.к. метод `Dispose()` вызывается пользователем объекта, а не сборщиком мусора. Освобождаемые структуры `ref` обсуждались в главе 4.

В целях иллюстрации применения интерфейса `IDisposable` создайте новый проект консольного приложения C# по имени `SimpleDispose`. Ниже приведен модифицированный класс `MyResourceWrapper`, который вместо переопределения метода `System.Object.Finalize()` теперь реализует интерфейс `IDisposable`:

```
using System;
namespace SimpleDispose
{
  // Реализация интерфейса IDisposable.
  class MyResourceWrapper : IDisposable
  {
```

```
    // После окончания работы с объектом пользователь
    // объекта должен вызывать этот метод.
    public void Dispose()
    {
      // Очистить неуправляемые ресурсы...
      // Освободить другие освобождаемые объекты, содержащиеся внутри.
      // Только для целей тестирования.
      Console.WriteLine("***** In Dispose! *****");
    }
  }
}
```

Обратите внимание, что метод `Dispose()` отвечает не только за освобождение неуправляемых ресурсов самого типа, но может также вызывать методы `Dispose()` для любых других освобождаемых объектов, которые содержатся внутри типа. В отличие от `Finalize()` в методе `Dispose()` вполне безопасно взаимодействовать с другими управляемыми объектами. Причина проста: сборщик мусора не имеет понятия об интерфейсе `IDisposable`, а потому никогда не будет вызывать метод `Dispose()`. Следовательно, когда пользователь объекта вызывает данный метод, объект все еще существует в управляемой куче и имеет доступ ко всем остальным находящимся там объектам. Логика вызова метода `Dispose()` прямолинейна:

```
using System;
using System.IO;
using SimpleDispose;
Console.WriteLine("***** Fun with Dispose *****\n");
// Создать освобождаемый объект и вызвать метод Dispose()
// для освобождения любых внутренних ресурсов.
MyResourceWrapper rw = new MyResourceWrapper();
rw.Dispose();
Console.ReadLine();
```

Конечно, перед попыткой вызова метода `Dispose()` на объекте понадобится проверить, поддерживает ли тип интерфейс `IDisposable`. Хотя всегда можно выяснить, какие типы в библиотеках базовых классов реализуют `IDisposable`, заглянув в документацию, программная проверка производится с помощью ключевого слова `is` или `as` (см. главу 6):

```
Console.WriteLine("***** Fun with Dispose *****\n");
MyResourceWrapper rw = new MyResourceWrapper();
if (rw is IDisposable)
{
  rw.Dispose();
}
Console.ReadLine();
```

Приведенный пример раскрывает очередное правило, касающееся управления памятью.

Правило. Неплохо вызывать метод `Dispose()` на любом создаваемом напрямую объекте, если он поддерживает интерфейс `IDisposable`. Предположение заключается в том, что когда проектировщик типа решил реализовать метод `Dispose()`, тогда тип должен выполнять какую-то очистку. Если вы забудете вызвать `Dispose()`, то память в конечном итоге будет очищена (так что можно не переживать), но это может занять больше времени, чем необходимо.

С предыдущим правилом связано одно предостережение. Несколько типов в библиотеках базовых классов, которые реализуют интерфейс IDisposable, предоставляют (кое в чем сбивающий с толку) псевдоним для метода Dispose() в попытке сделать имя метода очистки более естественным для определяющего его типа. В качестве примера можно взять класс System.IO.FileStream, который реализует интерфейс IDisposable (и потому поддерживает метод Dispose()), но также определяет следующий метод Close(), предназначенный для той же цели:

```
// Предполагается, что было импортировано пространство имен System.IO.
static void DisposeFileStream()
{
  FileStream fs = new FileStream("myFile.txt", FileMode.OpenOrCreate);

  // Мягко выражаясь, сбивает с толку!
  // Вызовы этих методов делают одно и то же!
  fs.Close();
  fs.Dispose();
}
```

В то время как "закрытие" (close) файла выглядит более естественным, чем его "освобождение" (dispose), подобное дублирование методов очистки может запутывать. При работе с типами, предлагающими псевдонимы, просто помните о том, что если тип реализует интерфейс IDisposable, то вызов метода Dispose() всегда является безопасным способом действия.

Повторное использование ключевого слова using в C#

Имея дело с управляемым объектом, который реализует интерфейс IDisposable, довольно часто приходится применять структурированную обработку исключений, гарантируя тем самым, что метод Dispose() типа будет вызываться даже в случае генерации исключения во время выполнения:

```
Console.WriteLine("***** Fun with Dispose *****\n");
MyResourceWrapper rw = new MyResourceWrapper ();
try
{
  // Использовать члены rw.
}
finally
{
  // Всегда вызывать Dispose(), возникла ошибка или нет.
  rw.Dispose();
}
```

Хотя это является хорошим примером защитного программирования, в действительности лишь немногих разработчиков привлекает перспектива помещения каждого освобождаемого типа внутрь блока try/finally, просто чтобы гарантировать вызов метода Dispose(). Того же самого результата можно достичь гораздо менее навязчивым способом, используя специальный фрагмент синтаксиса C#, который выглядит следующим образом:

```
Console.WriteLine("***** Fun with Dispose *****\n");
// Метод Dispose() вызывается автоматически
// при выходе за пределы области действия using.
```

```
using(MyResourceWrapper rw = new MyResourceWrapper())
{
  // Использовать объект rw.
}
```

Если вы просмотрите код CIL операторов верхнего уровня посредством `ildasm.exe`, то обнаружите, что синтаксис `using` на самом деле расширяется до логики `try/finally` с вполне ожидаемым вызовом `Dispose()`:

```
.method private hidebysig static void
  '<Main>$'(string[] args) cil managed
{
  ...
  .try
  {
  } // end .try
  finally
  {
    IL_0019: callvirt
        instance void [System.Runtime]System.IDisposable::Dispose()
  } // end handler
} // end of method '<Program>$'::'<Main>$'
```

На заметку! Попытка применения `using` к объекту, который не реализует интерфейс `IDisposable`, приводит к ошибке на этапе компиляции.

Несмотря на то что такой синтаксис устраняет необходимость вручную помещать освобождаемые объекты внутрь блоков `try/finally`, к сожалению, теперь ключевое слово `using` в C# имеет двойной смысл (импортирование пространств имен и вызов метода `Dispose()`). Однако при работе с типами, которые поддерживают интерфейс `IDisposable`, такая синтаксическая конструкция будет гарантировать, что используемый объект автоматический вызовет свой метод `Dispose()` по завершении блока `using`.

Кроме того, имейте в виду, что внутри `using` допускается объявлять несколько объектов *одного и того же типа*. Как и можно было ожидать, компилятор вставит код для вызова `Dispose()` на каждом объявленном объекте:

```
// Использовать список с разделителями-запятыми для объявления
// нескольких объектов, подлежащих освобождению.
using(MyResourceWrapper rw = new MyResourceWrapper(),
                        rw2 = new MyResourceWrapper())
{
  // Работать с объектами rw и rw2.
}
```

Объявления `using` (нововведение в версии 8.0)

В версии C# 8.0 были добавлены *объявления `using`*. Объявление `using` представляет собой объявление переменной, предваренное ключевым словом `using`. Функциональность объявления `using` будет такой же, как у синтаксиса, описанного в предыдущем разделе, за исключением явного блока кода, помещенного внутрь фигурных скобок (`{}`).

Добавьте к своему классу следующий метод:

```
private static void UsingDeclaration()
{
  // Эта переменная будет находиться в области видимости
  // вплоть до конца метода.
  using var rw = new MyResourceWrapper();
  // Сделать что-нибудь.
  Console.WriteLine("About to dispose.");
  // В этой точке переменная освобождается.
}
```

Далее добавьте к своим операторам верхнего уровня показанный ниже вызов:

```
Console.WriteLine("***** Fun with Dispose *****\n");
...
Console.WriteLine("Demonstrate using declarations");
UsingDeclaration();
Console.ReadLine();
```

Если вы изучите новый метод с помощью `ildasm.exe`, то (вполне ожидаемо) обнаружите тот же код, что и ранее:

```
.method private hidebysig static
  void UsingDeclaration() cil managed
{
  ...
  .try
  {
    ...
  } // end .try
  finally
  {
    IL_0018: callvirt instance void
      [System.Runtime]System.IDisposable::Dispose()
    ...
  } // end handler
  IL_001f: ret
} // end of method Program::UsingDeclaration
```

По сути, это новое средство является "магией" компилятора, позволяющей сэкономить несколько нажатий клавиш. При его использовании соблюдайте осторожность, т.к. новый синтаксис не настолько ясен, как предыдущий.

Создание финализируемых и освобождаемых типов

К настоящему моменту вы видели два разных подхода к конструированию класса, который очищает внутренние неуправляемые ресурсы. С одной стороны, можно применять финализатор. Использование такого подхода дает уверенность в том, что объект будет очищать себя сам во время сборки мусора (когда бы она ни произошла) без вмешательства со стороны пользователя. С другой стороны, можно реализовать интерфейс `IDisposable` и предоставить пользователю объекта способ очистки объекта по окончании работы с ним. Тем не менее, если пользователь объекта забудет вызвать метод `Dispose()`, то неуправляемые ресурсы могут оставаться в памяти неопределенно долго.

Нетрудно догадаться, что в одном определении класса можно смешивать оба подхода, извлекая лучшее из обеих моделей. Если пользователь объекта не забыл вызвать метод `Dispose()`, тогда можно проинформировать сборщик мусора о пропуске процесса финализации, вызвав метод `GC.SuppressFinalize()`. Если же пользователь объекта забыл вызвать `Dispose()`, то объект со временем будет финализирован и получит шанс освободить внутренние ресурсы. Преимущество здесь в том, что внутренние неуправляемые ресурсы будут тем или иным способом освобождены.

Ниже представлена очередная версия класса `MyResourceWrapper`, который теперь является финализируемым и освобождаемым; она определена в проекте консольного приложения C# по имени `FinalizableDisposableClass`:

```
using System;

namespace FinalizableDisposableClass
{
  // Усовершенствованная оболочка для ресурсов.
  public class MyResourceWrapper : IDisposable
  {
    // Сборщик мусора будет вызывать этот метод, если
    // пользователь объекта забыл вызвать Dispose().
    ~MyResourceWrapper()
    {
      // Очистить любые внутренние неуправляемые ресурсы.
      // **Не** вызывать Dispose() на управляемых объектах.
    }
    // Пользователь объекта будет вызывать этот метод
    // для как можно более скорой очистки ресурсов.
    public void Dispose()
    {
      // Очистить неуправляемые ресурсы.
      // Вызвать Dispose() для других освобождаемых объектов,
      // содержащихся внутри.
      // Если пользователь вызвал Dispose(), то финализация
      // не нужна, поэтому подавить ее.
      GC.SuppressFinalize(this);
    }
  }
}
```

Обратите внимание, что метод `Dispose()` был модифицирован для вызова метода `GC.SuppressFinalize()`, который информирует исполняющую среду о том, что вызывать деструктор при обработке данного объекта сборщиком мусора больше не обязательно, т.к. неуправляемые ресурсы уже освобождены посредством логики `Dispose()`.

Формализованный шаблон освобождения

Текущая реализация класса `MyResourceWrapper` работает довольно хорошо, но осталось еще несколько небольших недостатков. Во-первых, методы `Finalize()` и `Dispose()` должны освобождать те же самые неуправляемые ресурсы. Это может привести к появлению дублированного кода, что существенно усложнит сопровождение. В идеале следовало бы определить закрытый вспомогательный метод и вызывать его внутри указанных методов.

Во-вторых, желательно удостовериться в том, что метод `Finalize()` не пытается освободить любые управляемые объекты, когда такие действия должен делать метод `Dispose()`. В-третьих, имеет смысл также позаботиться о том, чтобы пользователь объекта мог безопасно вызывать метод `Dispose()` много раз без возникновения ошибки. В настоящий момент защита подобного рода в методе `Dispose()` отсутствует.

Для решения таких проектных задач в Microsoft определили формальный шаблон освобождения, который соблюдает баланс между надежностью, удобством сопровождения и производительностью. Вот окончательная версия класса `MyResourceWrapper`, в которой применяется официальный шаблон:

```
class MyResourceWrapper : IDisposable
{
  // Используется для выяснения, вызывался ли метод Dispose().
  private bool disposed = false;

  public void Dispose()
  {
    // Вызвать вспомогательный метод.
    // Указание true означает, что очистку
    // запустил пользователь объекта.
    CleanUp(true);

    // Подавить финализацию.
    GC.SuppressFinalize(this);
  }

  private void CleanUp(bool disposing)
  {
    // Удостовериться, не выполнялось ли уже освобождение.
    if (!this.disposed)
    {
      // Если disposing равно true, тогда
      // освободить все управляемые ресурсы.
      if (disposing)
      {
        // Освободить управляемые ресурсы.
      }
      // Очистить неуправляемые ресурсы.
    }
    disposed = true;
  }
  ~MyResourceWrapper()
  {
    // Вызвать вспомогательный метод.
    // Указание false означает, что
    // очистку запустил сборщик мусора.
    CleanUp(false);
  }
}
```

Обратите внимание, что в `MyResourceWrapper` теперь определен закрытый вспомогательный метод по имени `CleanUp()`. Передавая ему `true` в качестве аргумента, мы указываем, что очистку инициировал пользователь объекта, поэтому должны быть очищены все управляемые *и* неуправляемые ресурсы. Однако когда очистка иниции-

руется сборщиком мусора, при вызове методу `CleanUp()` передается значение `false`, чтобы внутренние освобождаемые объекты *не* освобождались (поскольку нельзя рассчитывать на то, что они все еще присутствуют в памяти). И, наконец, перед выходом из `CleanUp()` переменная-член `disposed` типа `bool` устанавливается в `true`, что дает возможность вызывать метод `Dispose()` много раз без возникновения ошибки.

На заметку! После того как объект был "освобожден", клиент по-прежнему может обращаться к его членам, т.к. объект пока еще находится в памяти. Следовательно, в надежном классе оболочки для ресурсов каждый член также необходимо снабдить дополнительной логикой, которая бы сообщала: "если объект освобожден, то ничего не делать, а просто возвратить управление".

Чтобы протестировать финальную версию класса `MyResourceWrapper`, модифицируйте свой файл `Program.cs`, как показано ниже:

```
using System;
using FinalizableDisposableClass;

Console.WriteLine("***** Dispose() / Destructor Combo Platter *****");

// Вызвать метод Dispose() вручную, что не приводит к вызову финализатора.
MyResourceWrapper rw = new MyResourceWrapper();
rw.Dispose();

// Не вызывать метод Dispose(). Это запустит финализатор,
// когда объект будет обрабатываться сборщиком мусора.
MyResourceWrapper rw2 = new MyResourceWrapper();
```

В коде явно вызывается метод `Dispose()` на объекте `rw`, поэтому вызов деструктора подавляется. Тем не менее, мы "забыли" вызвать метод `Dispose()` на объекте `rw2`; переживать не стоит — финализатор все равно выполнится при обработке объекта сборщиком мусора.

На этом исследование особенностей управления объектами со стороны исполняющей среды через сборку мусора завершено. Хотя дополнительные (довольно экзотические) детали, касающиеся процесса сборки мусора (такие как слабые ссылки и восстановление объектов), здесь не рассматривались, полученных сведений должно быть вполне достаточно, чтобы продолжить изучение самостоятельно. В завершение главы мы взглянем на программное средство под названием *ленивое (отложенное) создание* объектов.

Ленивое создание объектов

При создании классов иногда приходится учитывать, что отдельная переменная-член на самом деле может никогда не понадобиться из-за того, что пользователь объекта не будет обращаться к методу (или свойству), в котором она используется. Действительно, подобное происходит нередко. Однако проблема может возникнуть, если создание такой переменной-члена сопряжено с выделением большого объема памяти.

В качестве примера предположим, что строится класс, который инкапсулирует операции цифрового музыкального проигрывателя. В дополнение к ожидаемым методам вроде `Play()`, `Pause()` и `Stop()` вы также хотите обеспечить возможность возвращения коллекции объектов `Song` (посредством класса по имени `AllTracks`), которая представляет все имеющиеся на устройстве цифровые музыкальные файлы.

Создайте новый проект консольного приложения по имени LazyObjectInstantiation и определите в нем следующие классы:

```
// Song.cs
namespace LazyObjectInstantiation
{
  // Представляет одиночную композицию.
  class Song
  {
    public string Artist { get; set; }
    public string TrackName { get; set; }
    public double TrackLength { get; set; }
  }
}
// AllTracks.cs
using System;
namespace LazyObjectInstantiation
{
  // Представляет все композиции в проигрывателе.
  class AllTracks
  {
    // Наш проигрыватель может содержать
    // максимум 10 000 композиций.
    private Song[] _allSongs = new Song[10000];

    public AllTracks()
    {
      // Предположим, что здесь производится
      // заполнение массива объектов Song.
      Console.WriteLine("Filling up the songs!");
    }
  }
}
// MediaPlayer.cs
using System;
namespace LazyObjectInstantiation
{
  // Объект MediaPlayer имеет объекты AllTracks.
  class MediaPlayer
  {
    // Предположим, что эти методы делают что-то полезное.
    public void Play()  { /* Воспроизведение композиции*/ }
    public void Pause() { /* Пауза в воспроизведении  */ }
    public void Stop()  { /* Останов воспроизведения  */ }
    private AllTracks _allSongs = new AllTracks();

    public AllTracks GetAllTracks()
    {
      // Возвратить все композиции.
      return _allSongs;
    }
  }
}
```

В текущей реализации MediaPlayer предполагается, что пользователь объекта пожелает получать список объектов с помощью метода GetAllTracks(). Хорошо, а что если пользователю объекта такой список *не* нужен? В этой реализации память под переменную-член AllTracks по-прежнему будет выделяться, приводя тем самым к созданию 10 000 объектов Song в памяти:

```
using System;
using LazyObjectInstantiation;

Console.WriteLine("***** Fun with Lazy Instantiation *****\n");

// В этом вызывающем коде получение всех композиций не производится,
// но косвенно все равно создаются 10 000 объектов!
MediaPlayer myPlayer = new MediaPlayer();
myPlayer.Play();

Console.ReadLine();
```

Безусловно, лучше не создавать 10 000 объектов, с которыми никто не будет работать, потому что в результате нагрузка на сборщик мусора .NET Core намного увеличится. В то время как можно вручную добавить код, который обеспечит создание объекта _allSongs только в случае, если он применяется (скажем, используя шаблон фабричного метода), есть более простой путь.

Библиотеки базовых классов предоставляют удобный обобщенный класс по имени Lazy<>, который определен в пространстве имен System внутри сборки mscorlib.dll. Он позволяет определять данные, которые *не* будут создаваться до тех пор, пока действительно не начнут применяться в коде. Поскольку класс является обобщенным, при первом его использовании вы должны явно указать тип создаваемого элемента, которым может быть любой тип из библиотек базовых классов .NET Core или специальный тип, построенный вами самостоятельно. Чтобы включить отложенную инициализацию переменной-члена AllTracks, просто приведите код MediaPlayer к следующему виду:

```
// Объект MediaPlayer имеет объект Lazy<AllTracks>.
class MediaPlayer
{
  ...
  private Lazy<AllTracks> _allSongs = new Lazy<AllTracks>();
  public AllTracks GetAllTracks()
  {
    // Возвратить все композиции.
    return _allSongs.Value;
  }
}
```

Помимо того факта, что переменная-член AllTracks теперь имеет тип Lazy<>, важно обратить внимание на изменение также и реализации показанного выше метода GetAllTracks(). В частности, для получения актуальных сохраненных данных (в этом случае объекта AllTracks, поддерживающего 10 000 объектов Song) должно применяться доступное только для чтения свойство Value класса Lazy<>.

Взгляните, как благодаря такому простому изменению приведенный далее модифицированный код будет косвенно размещать объекты Song в памяти, только если метод GetAllTracks() действительно вызывается:

```
Console.WriteLine("***** Fun with Lazy Instantiation *****\n");

// Память под объект AllTracks здесь не выделяется!
MediaPlayer myPlayer = new MediaPlayer();
myPlayer.Play();

// Размещение объекта AllTracks происходит
// только в случае вызова метода GetAllTracks().
MediaPlayer yourPlayer = new MediaPlayer();
AllTracks yourMusic = yourPlayer.GetAllTracks();

Console.ReadLine();
```

На заметку! Ленивое создание объектов полезно не только для уменьшения количества выделений памяти под ненужные объекты. Этот прием можно также использовать в ситуации, когда для создания члена применяется затратный в плане ресурсов код, такой как вызов удаленного метода, взаимодействие с реляционной базой данных и т.п.

Настройка процесса создания данных Lazy<>

При объявлении переменной Lazy<> действительный внутренний тип данных создается с использованием стандартного конструктора:

```
// При использовании переменной Lazy<> вызывается
// стандартный конструктор класса AllTracks.
private Lazy<AllTracks> _allSongs = new Lazy<AllTracks>();
```

В некоторых случаях приведенный код может оказаться приемлемым, но что если класс AllTracks имеет дополнительные конструкторы и нужно обеспечить вызов подходящего конструктора? Более того, что если при создании переменной Lazy() должна выполняться какая-то специальная работа (кроме простого создания объекта AllTracks)? К счастью, класс Lazy() позволяет указывать в качестве необязательного параметра обобщенный делегат, который задает метод для вызова во время создания находящегося внутри типа.

Таким обобщенным делегатом является тип System.Func<>, который может указывать на метод, возвращающий тот же самый тип данных, что и создаваемый связанной переменной Lazy<>, и способный принимать вплоть до 16 аргументов (типизированных с применением обобщенных параметров типа). В большинстве случаев никаких параметров для передачи методу, на который указывает Func<>, задавать не придется. Вдобавок, чтобы значительно упростить работу с типом Func<>, рекомендуется использовать лямбда-выражения (отношения между делегатами и лямбда-выражениями подробно освещаются в главе 12).

Ниже показана окончательная версия класса MediaPlayer, в которой добавлен небольшой специальный код, выполняемый при создании внутреннего объекта AllTracks. Не забывайте, что перед завершением метод должен возвратить новый экземпляр типа, помещенного в Lazy<>, причем применять можно любой конструктор по своему выбору (здесь по-прежнему вызывается стандартный конструктор AllTracks).

```
class MediaPlayer
{
  ...
  // Использовать лямбда-выражение для добавления дополнительного
  // кода, который выполняется при создании объекта AllTracks.
  private Lazy<AllTracks> _allSongs =
    new Lazy<AllTracks>( () =>
    {
      Console.WriteLine("Creating AllTracks object!");
      return new AllTracks();
    }
  );
  public AllTracks GetAllTracks()
  {
    // Возвратить все композиции.
    return _allSongs.Value;
  }
}
```

Итак, вы наверняка смогли оценить полезность класса `Lazy<>`. По существу этот обобщенный класс позволяет гарантировать, что затратные в плане ресурсов объекты размещаются в памяти, только когда они требуются их пользователю.

Резюме

Целью настоящей главы было прояснение процесса сборки мусора. Вы видели, что сборщик мусора запускается, только если не удается получить необходимый объем памяти из управляемой кучи (либо когда разработчик вызывает `GC.Collect()`). Не забывайте о том, что разработанный в Microsoft алгоритм сборки мусора хорошо оптимизирован и предусматривает использование поколений объектов, дополнительных потоков для финализации объектов и управляемой кучи для обслуживания крупных объектов.

В главе также было показано, каким образом программно взаимодействовать со сборщиком мусора с применением класса `System.GC`. Как отмечалось, единственным случаем, когда может возникнуть необходимость в подобном взаимодействии, является построение финализируемых или освобождаемых классов, которые имеют дело с неуправляемыми ресурсами.

Вспомните, что финализируемые типы — это классы, которые предоставляют деструктор (переопределяя метод `Finalize()`) для очистки неуправляемых ресурсов во время сборки мусора. С другой стороны, освобождаемые объекты являются классами (или структурами не `ref`), реализующими интерфейс `IDisposable`, к которому пользователь объекта должен обращаться по завершении работы с ними. Наконец, вы изучили официальный шаблон освобождения, в котором смешаны оба подхода.

В заключение был рассмотрен обобщенный класс по имени `Lazy<>`. Вы узнали, что данный класс позволяет отложить создание затратных (в смысле потребления памяти) объектов до тех пор, пока вызывающая сторона действительно не затребует их. Класс `Lazy<>` помогает сократить количество объектов, хранящихся в управляемой куче, и также обеспечивает создание затратных объектов только тогда, когда они действительно нужны в вызывающем коде.

ЧАСТЬ IV

Дополнительные конструкции программирования на C#

ГЛАВА 10

Коллекции и обобщения

Любому приложению, создаваемому с помощью платформы .NET Core, потребуется решать вопросы поддержки и манипулирования набором значений данных в памяти. Значения данных могут поступать из множества местоположений, включая реляционную базу данных, локальный текстовый файл, XML-документ, вызов веб-службы, или через предоставляемый пользователем источник ввода.

В первом выпуске платформы .NET программисты часто применяли классы из пространства имен `System.Collections` для хранения и взаимодействия с элементами данных, используемыми внутри приложения. В версии .NET 2.0 язык программирования C# был расширен поддержкой средства под названием *обобщения*, и вместе с этим изменением в библиотеках базовых классов появилось новое пространство имен — `System.Collections.Generic`.

В настоящей главе представлен обзор разнообразных пространств имен и типов коллекций (обобщенных и необобщенных), находящихся в библиотеках базовых классов .NET Core. Вы увидите, что обобщенные контейнеры часто превосходят свои необобщенные аналоги, поскольку они обычно обеспечивают лучшую безопасность в отношении типов и дают выигрыш в плане производительности. После того, как вы научитесь создавать и манипулировать обобщенными элементами внутри платформы, в оставшемся материале главы будет продемонстрировано создание собственных обобщенных методов и типов. Вы узнаете о роли *ограничений* (и соответствующего ключевого слова `where` языка C#), которые позволяют строить классы, в высшей степени безопасные в отношении типов.

Побудительные причины создания классов коллекций

Несомненно, самым элементарным контейнером, который допускается применять для хранения данных приложения, считается массив. В главе 4 вы узнали, что массив C# позволяет определить набор идентично типизированных элементов (в том числе массив элементов типа `System.Object`, по существу представляющий собой массив данных любых типов) с фиксированным верхним пределом. Кроме того, вспомните из главы 4, что все переменные массивов C# получают много функциональных возможностей от класса `System.Array`. В качестве краткого напоминания взгляните на следующий код, который создает массив текстовых данных и манипулирует его содержимым разными способами:

```
// Создать массив строковых данных.
string[] strArray = { "First", "Second", "Third" };
```

```
// Отобразить количество элементов в массиве с помощью свойства Length.
WriteLine("This array has {0} items.", strArray.Length);
Console.WriteLine();
// Отобразить содержимое массива, используя перечислитель.
foreach (string s in strArray)
{
  Console.WriteLine("Array Entry: {0}", s);
}
Console.WriteLine();
// Обратить массив и снова вывести его содержимое.
Array.Reverse(strArray);
foreach (string s in strArray)
{
  Console.WriteLine("Array Entry: {0}", s);
}
Console.ReadLine();
```

Хотя базовые массивы могут быть удобными для управления небольшими объемами данных фиксированного размера, есть немало случаев, когда требуются более гибкие структуры данных, такие как динамически расширяющийся и сокращающийся контейнер или контейнер, который может хранить только объекты, удовлетворяющие заданному критерию (например, объекты, производные от специфичного базового класса, или объекты, реализующие определенный интерфейс). Когда вы используете простой массив, всегда помните о том, что он был создан с "фиксированным размером". Если вы создали массив из трех элементов, то вы и получите только три элемента; следовательно, представленный далее код даст в результате исключение времени выполнения (конкретно — `IndexOutOfRangeException`):

```
// Создать массив строковых данных.
string[] strArray = { "First", "Second", "Third" };
// Попытка добавить новый элемент в конец массива?
// Ошибка во время выполнения!
strArray[3] = "new item?";
...
```

На заметку! На самом деле изменять размер массива можно с применением обобщенного метода `Resize<T>()`. Однако такое действие приведет к копированию данных в новый объект массива и может оказаться неэффективным.

Чтобы помочь в преодолении ограничений простого массива, библиотеки базовых классов .NET Core поставляются с несколькими пространствами имен, которые содержат *классы коллекций*. В отличие от простого массива C# классы коллекций построены с возможностью динамического изменения своих размеров на лету по мере вставки либо удаления из них элементов. Более того, многие классы коллекций предлагают улучшенную безопасность в отношении типов и всерьез оптимизированы для обработки содержащихся внутри данных в манере, эффективной с точки зрения затрат памяти. В ходе чтения главы вы быстро заметите, что класс коллекции может принадлежать к одной из двух обширных категорий:

- необобщенные коллекции (в основном находящиеся в пространстве имен `System.Collections`);
- обобщенные коллекции (в основном находящиеся в пространстве имен `System.Collections.Generic`).

Необобщенные коллекции обычно спроектированы для оперирования типами `System.Object` и, следовательно, являются слабо типизированными контейнерами (тем не менее, некоторые необобщенные коллекции работают только со специфическим типом данных наподобие объектов `string`). По контрасту обобщенные коллекции являются намного более безопасными в отношении типов, учитывая, что при создании вы должны указывать "вид типа" данных, которые они будут содержать. Как вы увидите, признаком любого обобщенного элемента является наличие "параметра типа", обозначаемого с помощью угловых скобок (например, `List<T>`). Детали обобщений (в том числе связанные с ними преимущества) будут исследоваться позже в этой главе. А сейчас давайте ознакомимся с некоторыми ключевыми типами необобщенных коллекций из пространств имен `System.Collections` и `System.Collections.Specialized`.

Пространство имен `System.Collections`

С самого первого выпуска платформы .NET программисты часто использовали классы необобщенных коллекций из пространства имен `System.Collecitons`, которое содержит набор классов, предназначенных для управления и организации крупных объемов данных в памяти. В табл. 10.1 документированы распространенные классы коллекций, определенные в этом пространстве имен, а также основные интерфейсы, которые они реализуют.

Таблица 10.1. Полезные классы из пространства имен `System.Collections`

Класс `System.Collections`	Описание	Основные реализуемые интерфейсы
`ArrayList`	Представляет коллекцию с динамически изменяемым размером, выдающую объекты в последовательном порядке	`IList`, `ICollection`, `IEnumerable` и `ICloneable`
`BitArray`	Управляет компактным массивом битовых значений, которые представляются как булевские, где `true` обозначает установленный (1) бит, а `false` — неустановленный (0) бит	`ICollection`, `IEnumerable` и `ICloneable`
`Hashtable`	Представляет коллекцию пар "ключ-значение", организованных на основе хеш-кода ключа	`IDictionary`, `ICollection`, `IEnumerable` и `ICloneable`
`Queue`	Представляет стандартную очередь объектов, работающую по принципу FIFO ("первый вошел — первый вышел")	`ICollection`, `IEnumerable` и `ICloneable`
`SortedList`	Представляет коллекцию пар "ключ-значение", отсортированных по ключу и доступных по ключу и по индексу	`IDictionary`, `ICollection`, `IEnumerable` и `ICloneable`
`Stack`	Представляет стек LIFO ("последний вошел — первый вышел"), поддерживающий функциональность заталкивания и выталкивания, а также считывания	`ICollection`, `IEnumerable` и `ICloneable`

Интерфейсы, реализованные перечисленными в табл. 10.1 классами коллекций, позволяют проникнуть в суть их общей функциональности. В табл. 10.2 представлено описание общей природы основных интерфейсов, часть из которых кратко обсуждалась в главе 8.

Таблица 10.2. Основные интерфейсы, поддерживаемые классами из пространства имен `System.Collections`

Интерфейс `System.Collections`	Описание
`ICollection`	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех необобщенных типов коллекций
`ICloneable`	Позволяет реализующему объекту возвращать вызывающему коду копию самого себя
`IDictionary`	Позволяет объекту необобщенной коллекции представлять свое содержимое в виде пар "ключ-значение"
`IEnumerable`	Возвращает объект, реализующий интерфейс `IEnumerator` (см. следующую строку в таблице)
`IEnumerator`	Делает возможной итерацию в стиле `foreach` по элементам коллекции
`IList`	Обеспечивает поведение добавления, удаления и индексирования элементов в последовательном списке объектов

Иллюстративный пример: работа с ArrayList

Возможно, вы уже имеете начальный опыт применения (или реализации) некоторых из указанных выше классических структур данных, таких как стеки, очереди или списки. Если это не так, то при рассмотрении обобщенных аналогов таких структур позже в главе будут предоставлены дополнительные сведения об отличиях между ними. А пока что взгляните на пример кода, в котором используется объект `ArrayList`:

```
// Для доступа к ArrayList потребуется импортировать
// пространство имен System.Collections.
using System.Collections;
ArrayList strArray = new ArrayList();
strArray.AddRange(new string[] { "First", "Second", "Third" });

// Отобразить количество элементов в ArrayList.
System.Console.WriteLine("This collection has {0} items.", strArray.Count);
System.Console.WriteLine();

// Добавить новый элемент и отобразить текущее их количество.
strArray.Add("Fourth!");
System.Console.WriteLine("This collection has {0} items.", strArray.Count);

// Отобразить содержимое.
foreach (string s in strArray)
{
  System.Console.WriteLine("Entry: {0}", s);
}
System.Console.WriteLine();
```

Обратите внимание, что вы можете добавлять (и удалять) элементы на лету, а контейнер автоматически будет соответствующим образом изменять свой размер.

Как вы могли догадаться, помимо свойства `Count` и методов `AddRange()` и `Add()` класс `ArrayList` имеет много полезных членов, которые полностью описаны в документации по .NET Core. К слову, другие классы `System.Collections` (`Stack`, `Queue` и т.д.) тоже подробно документированы в справочной системе .NET Core.

Однако важно отметить, что в большинстве ваших проектов .NET Core классы коллекций из пространства имен `System.Collections`, скорее всего, применяться не будут! В наши дни намного чаще используются их обобщенные аналоги, находящиеся в пространстве имен `System.Collections.Generic`. С учетом сказанного остальные необобщенные классы из `System.Collections` здесь не обсуждаются (и примеры работы с ними не приводятся).

Обзор пространства имен `System.Collections.Specialized`

`System.Collections` — не единственное пространство имен .NET Core, которое содержит необобщенные классы коллекций. В пространстве имен `System.Collections.Specialized` определено несколько специализированных типов коллекций. В табл. 10.3 описаны наиболее полезные типы в этом конкретном пространстве имен, которые все являются необобщенными.

Таблица 10.3. Полезные классы из пространства имен `System.Collections.Specialized`

Класс `System.Collections.Specialized`	Описание
`HybridDictionary`	Этот класс реализует интерфейс `IDictionary` за счет применения `ListDictionary`, пока коллекция мала, и переключения на `Hashtable`, когда коллекция становится большой
`ListDictionary`	Этот класс удобен, когда необходимо управлять небольшим количеством элементов (10 или около того), которые могут изменяться с течением времени. Для управления своими данными класс использует односвязный список
`StringCollection`	Этот класс обеспечивает оптимальный способ для управления крупными коллекциями строковых данных
`BitVector32`	Этот класс предоставляет простую структуру, которая хранит булевские значения и небольшие целые числа в 32 битах памяти

Кроме указанных конкретных типов классов пространство имен `System.Collections.Specialized` также содержит много дополнительных интерфейсов и абстрактных базовых классов, которые можно применять в качестве стартовых точек для создания специальных классов коллекций. Хотя в ряде ситуаций такие "специализированные" типы могут оказаться именно тем, что требуется в ваших проектах, здесь они рассматриваться не будут. И снова во многих ситуациях вы с высокой вероятностью обнаружите, что пространство имен `System.Collections.Generic` предлагает классы с похожей функциональностью, но с добавочными преимуществами.

На заметку! В библиотеках базовых классов .NET Core доступны два дополнительных пространства имен, связанные с коллекциями (`System.Collections.ObjectModel` и `System.Collections.Concurrent`). Первое из них будет объясняться позже в главе, когда вы освоите тему обобщений. Пространство имен `System.Collections.Concurrent` предоставляет классы коллекций, хорошо подходящие для многопоточной среды (многопоточность обсуждается в главе 15).

Проблемы, присущие необобщенным коллекциям

Хотя на протяжении многих лет с использованием необобщенных классов коллекций (и интерфейсов) было построено немало успешных приложений .NET и .NET Core, опыт показал, что применение этих типов может привести к возникновению ряда проблем.

Первая проблема заключается в том, что использование классов коллекций `System.Collections` и `System.Collections.Specialized` в результате дает код с низкой производительностью, особенно в случае манипулирования числовыми данными (например, типами значений). Как вы вскоре увидите, когда структуры хранятся в любом необобщенном классе коллекции, прототипированном для оперирования с `System.Object`, среда CoreCLR должна осуществлять некоторое количество операций перемещения в памяти, что может нанести ущерб скорости выполнения.

Вторая проблема связана с тем, что большинство необобщенных классов коллекций не являются безопасными в отношении типов, т.к. они были созданы для работы с `System.Object` и потому могут содержать в себе вообще все что угодно. Если разработчик нуждался в создании безопасной в отношении типов коллекции (скажем, контейнера, который способен хранить объекты, реализующие только определенный интерфейс), то единственным реальным вариантом было создание нового класса коллекции вручную. Хотя задача не отличалась высокой трудоемкостью, решать ее было несколько утомительно.

Прежде чем вы увидите, как применять обобщения в своих программах, полезно чуть глубже рассмотреть недостатки необобщенных классов коллекций, что поможет лучше понять проблемы, которые был призван решить механизм обобщений. Создайте новый проект консольного приложения по имени `IssuesWithNongenericCollections`, импортируйте пространства имен `System` и `System.Collections` в начале файла `Program.cs` и удалите оставшийся код:

```
using System;
using System.Collections;
```

Проблема производительности

Как уже было указано в главе 4, платформа .NET Core поддерживает две обширные категории данных: типы значений и ссылочные типы. Поскольку в .NET Core определены две основные категории типов, временами возникает необходимость представить переменную одной категории как переменную другой категории. Для этого в C# предлагается простой механизм, называемый *упаковкой* (boxing), который позволяет хранить данные типа значения внутри ссылочной переменной. Предположим, что в методе по имени `SimpleBoxUnboxOperation()` создана локальная переменная типа `int`. Если где-то в приложении понадобится представить такой тип значения как ссылочный тип, то значение придется *упаковать*:

```
static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;
}
```

Упаковку можно формально определить как процесс явного присваивания данных типа значения переменной `System.Object`. При упаковке значения среда CoreCLR размещает в куче новый объект и копирует в него величину типа значения (в данном случае 25). В качестве результата возвращается ссылка на вновь размещенный в куче объект.

Противоположная операция также разрешена и называется *распаковкой* (unboxing). Распаковка представляет собой процесс преобразования значения, хранящегося в объектной ссылке, обратно в соответствующий тип значения в стеке. Синтаксически операция распаковки выглядит как обычная операция приведения, но ее семантика несколько отличается. Среда CoreCLR начинает с проверки того, что полученный тип данных эквивалентен упакованному типу, и если это так, то копирует значение в переменную, находящуюся в стеке. Например, следующие операции распаковки работают успешно при условии, что лежащим в основе типом `boxedInt` действительно является `int`:

```
static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;

  // Распаковать ссылку обратно в int.
  int unboxedInt = (int)boxedInt;
}
```

Когда компилятор C# встречает синтаксис упаковки/распаковки, он выпускает код CIL, который содержит коды операций `box`/`unbox`. Если вы просмотрите сборку с помощью утилиты `ildasm.exe`, то обнаружите в ней показанный далее код CIL:

```
.method assembly hidebysig static
  void  '<<Main>$>g__SimpleBoxUnboxOperation|0_0'() cil managed
{
  .maxstack 1
  .locals init (int32 V_0, object V_1, int32 V_2)
  IL_0000:  nop
  IL_0001:  ldc.i4.s   25
  IL_0003:  stloc.0
  IL_0004:  ldloc.0
  IL_0005:  box        [System.Runtime]System.Int32
  IL_000a:  stloc.1
  IL_000b:  ldloc.1
  IL_000c:  unbox.any  [System.Runtime]System.Int32
  IL_0011:  stloc.2
  IL_0012:  ret
} // end of method '<Program>$'::'<<Main>$>g__SimpleBoxUnboxOperation|0_0'
```

Помните, что в отличие от обычного приведения распаковка *обязана* осуществляться только в подходящий тип данных. Попытка распаковать порцию данных в некорректный тип приводит к генерации исключения `InvalidCastException`. Для обеспечения высокой безопасности каждая операция распаковки должна быть помещена внутрь конструкции `try/catch`, но такое действие со всеми операциями распаковки в приложении может оказаться достаточно трудоемкой задачей. Ниже показан измененный код, который выдаст ошибку из-за того, что в нем предпринята попытка распаковки упакованного значения `int` в тип `long`:

```
static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;

  // Распаковать в неподходящий тип данных, чтобы
  // инициировать исключение времени выполнения.
  try
  {
    long unboxedLong = (long)boxedInt;
  }
  catch (InvalidCastException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

На первый взгляд упаковка/распаковка может показаться довольно непримечательным средством языка, с которым связан больше академический интерес, нежели практическая ценность. В конце концов, необходимость хранения локального типа значения в локальной переменной `object` будет возникать нечасто. Тем не менее, оказывается, что процесс упаковки/распаковки очень полезен, поскольку позволяет предполагать, что все можно трактовать как `System.Object`, а среда CoreCLR самостоятельно позаботится о деталях, касающихся памяти.

Давайте обратимся к практическому применению описанных приемов. Мы будем исследовать класс `System.Collections.ArrayList` и использовать его для хранения порции числовых (расположенных в стеке) данных. Соответствующие члены класса `ArrayList` перечислены ниже. Обратите внимание, что они прототипированы для работы с данными типа `System.Object`. Теперь рассмотрим методы `Add()`, `Insert()` и `Remove()`, а также индексатор класса:

```
public class ArrayList : IList, ICloneable
{
  ...
  public virtual int Add(object? value);
  public virtual void Insert(int index, object? value);
  public virtual void Remove(object? obj);
  public virtual object? this[int index] { get; set; }
}
```

Класс `ArrayList` был построен для оперирования с экземплярами `object`, которые представляют данные, находящиеся в куче, поэтому может показаться странным, что следующий код компилируется и выполняется без ошибок:

```
static void WorkWithArrayList()
{
  // Типы значений автоматически упаковываются при передаче
  // методу, который требует экземпляр типа object.
  ArrayList myInts = new ArrayList();
  myInts.Add(10);
  myInts.Add(20);
  myInts.Add(35);
}
```

Хотя здесь числовые данные напрямую передаются методам, которые требуют экземпляров типа `object`, исполняющая среда выполняет автоматическую упаковку таких основанных на стеке данных. Когда позже понадобится извлечь элемент из `ArrayList` с применением индексатора типа, находящийся в куче объект должен быть распакован в целочисленное значение, расположенное в стеке, посредством операции приведения. Не забывайте, что индексатор `ArrayList` возвращает элементы типа `System.Object`, а не `System.Int32`:

```
static void WorkWithArrayList()
{
  // Типы значений автоматически упаковываются,
  // когда передаются члену, принимающему object.
  ArrayList myInts = new ArrayList();
  myInts.Add(10);
  myInts.Add(20);
  myInts.Add(35);

  // Распаковка происходит, когда объект преобразуется
  // обратно в данные, расположенные в стеке.
  int i = (int)myInts[0];

  // Теперь значение вновь упаковывается, т.к.
  // метод WriteLine() требует типа object!
  Console.WriteLine("Value of your int: {0}", i);
}
```

Обратите внимание, что расположенное в стеке значение типа `System.Int32` перед вызовом метода `ArrayList.Add()` упаковывается, чтобы оно могло быть передано в требуемом виде `System.Object`. Вдобавок объект `System.Object` распаковывается обратно в `System.Int32` после его извлечения из `ArrayList` через операцию приведения лишь для того, чтобы *снова* быть упакованными при передаче методу `Console.WriteLine()`, поскольку данный метод работает с типом `System.Object`.

Упаковка и распаковка удобны с точки зрения программиста, но такой упрощенный подход к передаче данных между стеком и кучей влечет за собой проблемы, связанные с производительностью (снижение скорости выполнения и увеличение размера кода), а также приводит к утрате безопасности в отношении типов. Чтобы понять проблемы с производительностью, примите во внимание действия, которые должны произойти при упаковке и распаковке простого целочисленного значения.

1. Новый объект должен быть размещен в управляемой куче.
2. Значение данных, находящееся в стеке, должно быть передано в выделенное место в памяти.
3. При распаковке значение, которое хранится в объекте, находящемся в куче, должно быть передано обратно в стек.

4. Неиспользуемый в дальнейшем объект, расположенный в куче, будет (со временем) удален сборщиком мусора.

Несмотря на то что показанный конкретный метод `WorkWithArrayList()` не создает значительное узкое место в плане производительности, вы определенно заметите такое влияние, если `ArrayList` будет содержать тысячи целочисленных значений, которыми программа манипулирует на регулярной основе. В идеальном мире мы могли бы обрабатывать данные, находящиеся внутри контейнера в стеке, безо всяких проблем с производительностью. Было бы замечательно иметь возможность извлекать данные из контейнера, не прибегая к конструкциям `try/catch` (именно это позволяют делать обобщения).

Проблема безопасности в отношении типов

Мы уже затрагивали проблему безопасности в отношении типов, когда рассматривали операции распаковки. Вспомните, что данные должны быть распакованы в тот же самый тип, с которым они объявлялись перед упаковкой. Однако существует еще один аспект безопасности в отношении типов, который необходимо иметь в виду в мире без обобщений: тот факт, что классы из пространства имен `System.Collections` обычно могут хранить любые данные, т.к. их члены прототипированы для оперирования с типом `System.Object`. Например, следующий метод строит список `ArrayList` с произвольными фрагментами несвязанных данных:

```
static void ArrayListOfRandomObjects()
{
  // ArrayList может хранить вообще все что угодно.
  ArrayList allMyObjects = new ArrayList();
  allMyObjects.Add(true);
  allMyObjects.Add(new OperatingSystem(PlatformID.MacOSX,
                                       new Version(10, 0)));
  allMyObjects.Add(66);
  allMyObjects.Add(3.14);
}
```

В ряде случаев вам будет требоваться исключительно гибкий контейнер, который способен хранить буквально все (как было здесь показано). Но большую часть времени вас интересует *безопасный в отношении типов* контейнер, который может работать только с определенным типом данных. Например, вы можете нуждаться в контейнере, хранящем только объекты типа подключения к базе данных, растрового изображения или класса, реализующего интерфейс `IPointy`.

До появления обобщений единственный способ решения проблемы, касающейся безопасности в отношении типов, предусматривал создание вручную специального класса (строго типизированной) коллекции. Предположим, что вы хотите создать специальную коллекцию, которая способна содержать только объекты типа `Person`:

```
namespace IssuesWithNonGenericCollections
{
  public class Person
  {
    public int Age {get; set;}
    public string FirstName {get; set;}
    public string LastName {get; set;}
```

```
    public Person(){}
    public Person(string firstName, string lastName, int age)
    {
      Age = age;
      FirstName = firstName;
      LastName = lastName;
    }
    public override string ToString()
    {
      return $"Name: {FirstName} {LastName}, Age: {Age}";
    }
  }
}
```

Чтобы построить коллекцию, которая способна хранить только объекты `Person`, можно определить переменную-член `System.Collection.ArrayList` внутри класса по имени `PeopleCollection` и сконфигурировать все члены для оперирования со строго типизированными объектами `Person`, а не с объектами типа `System.Object`. Ниже приведен простой пример (специальная коллекция производственного уровня могла бы поддерживать множество дополнительных членов и расширять абстрактный базовый класс из пространства имен `System.Collections` или `System.Collections.Specialized`):

```
using System.Collections;
namespace IssuesWithNonGenericCollections
{
  public class PersonCollection : IEnumerable
  {
    private ArrayList arPeople = new ArrayList();

    // Приведение для вызывающего кода.
    public Person GetPerson(int pos) => (Person)arPeople[pos];

    // Вставка только объектов Person.
    public void AddPerson(Person p)
    {
      arPeople.Add(p);
    }

    public void ClearPeople()
    {
      arPeople.Clear();
    }

    public int Count => arPeople.Count;

    // Поддержка перечисления с помощью foreach.
    IEnumerator IEnumerable.GetEnumerator() => arPeople.GetEnumerator();
  }
}
```

Обратите внимание, что класс `PeopleCollection` реализует интерфейс `IEnumerable`, который делает возможной итерацию в стиле `foreach` по всем элементам, содержащимся в коллекции. Кроме того, методы `GetPerson()` и `AddPerson()` прототипированы для работы только с объектами `Person`, а не растровыми изображениями, строками, подключениями к базам данных или другими элементами. Благодаря

определению таких классов теперь обеспечивается безопасность в отношении типов, учитывая, что компилятор C# будет способен выявить любую попытку вставки элемента несовместимого типа. Обновите операторы `using` в файле `Program.cs`, как показано ниже, и поместите в конец текущего кода метод `UserPersonCollection()`:

```
using System;
using System.Collections;
using IssuesWithNonGenericCollections;
// Операторы верхнего уровня в Program.cs.
static void UsePersonCollection()
{
  Console.WriteLine("***** Custom Person Collection *****\n");
  PersonCollection myPeople = new PersonCollection();
  myPeople.AddPerson(new Person("Homer", "Simpson", 40));
  myPeople.AddPerson(new Person("Marge", "Simpson", 38));
  myPeople.AddPerson(new Person("Lisa", "Simpson", 9));
  myPeople.AddPerson(new Person("Bart", "Simpson", 7));
  myPeople.AddPerson(new Person("Maggie", "Simpson", 2));

  // Это вызовет ошибку на этапе компиляции!
  // myPeople.AddPerson(new Car());

  foreach (Person p in myPeople)
  {
    Console.WriteLine(p);
  }
}
```

Хотя специальные коллекции гарантируют безопасность в отношении типов, такой подход обязывает создавать (в основном идентичные) специальные коллекции для всех уникальных типов данных, которые планируется в них помещать. Таким образом, если нужна специальная коллекция, которая могла бы оперировать только с классами, производными от базового класса `Car`, тогда придется построить очень похожий класс коллекции:

```
using System.Collections;
public class CarCollection : IEnumerable
{
  private ArrayList arCars = new ArrayList();

  // Приведение для вызывающего кода.
  public Car GetCar(int pos) => (Car)arCars[pos];

  // Вставка только объектов Car.
  public void AddCar(Car c)
  { arCars.Add(c); }

  public void ClearCars()
  { arCars.Clear(); }

  public int Count => arCars.Count;

  // Поддержка перечисления с помощью foreach.
  IEnumerator IEnumerable.GetEnumerator() => arCars.GetEnumerator();
}
```

Тем не менее, класс специальной коллекции ничего не делает для решения проблемы с накладными расходами по упаковке/распаковке. Даже если создать специаль-

ную коллекцию по имени IntCollection, которая предназначена для работы только с элементами System.Int32, то все равно придется выделять память под объект какого-нибудь вида, хранящий данные (например, System.Array и ArrayList):

```
public class IntCollection : IEnumerable
{
  private ArrayList arInts = new ArrayList();

  // Получение int (выполняется распаковка).
  public int GetInt(int pos) => (int)arInts[pos];

  // Вставка int (выполняется упаковка).
  public void AddInt(int i)
  {
    arInts.Add(i);
  }

  public void ClearInts()
  {
    arInts.Clear();
  }

  public int Count => arInts.Count;

  IEnumerator IEnumerable.GetEnumerator() => arInts.GetEnumerator();
}
```

Независимо от того, какой тип выбран для хранения целых чисел, в случае применения необобщенных контейнеров затруднительного положения с упаковкой избежать невозможно.

Первый взгляд на обобщенные коллекции

Когда используются классы обобщенных коллекций, все описанные выше проблемы исчезают, включая накладные расходы на упаковку/распаковку и отсутствие безопасности в отношении типов. К тому же необходимость в создании специального класса (обобщенной) коллекции становится довольно редкой. Вместо построения уникальных классов, которые могут хранить объекты людей, автомобилей и целые числа, можно задействовать класс обобщенной коллекции и указать тип хранимых элементов. Добавьте в начало файла Program.cs следующий оператор using:

```
using System.Collections.Generic;
```

Взгляните на показанный ниже метод (добавленный в конец файла Program.cs), в котором используется класс List<T> (из пространства имен System.Collection.Generic) для хранения разнообразных видов данных в строго типизированной манере (пока не обращайте внимания на детали синтаксиса обобщений):

```
static void UseGenericList()
{
  Console.WriteLine("***** Fun with Generics *****\n");

  // Этот объект List<> может хранить только объекты Person.
  List<Person> morePeople = new List<Person>();
  morePeople.Add(new Person ("Frank", "Black", 50));
  Console.WriteLine(morePeople[0]);

  // Этот объект List<> может хранить только целые числа.
  List<int> moreInts = new List<int>();
```

```
  moreInts.Add(10);
  moreInts.Add(2);
  int sum = moreInts[0] + moreInts[1];
  // Ошибка на этапе компиляции! Объект Person
  // не может быть добавлен в список элементов int!
  // moreInts.Add(new Person());
}
```

Первый контейнер `List<T>` способен содержать только объекты `Person`. По этой причине выполнять приведение при извлечении элементов из контейнера не требуется, что делает такой подход более безопасным в отношении типов. Второй контейнер `List<T>` может хранить только целые числа, размещенные в стеке; другими словами, здесь не происходит никакой скрытой упаковки/распаковки, которая имеет место в необобщенном типе `ArrayList`. Ниже приведен краткий перечень преимуществ обобщенных контейнеров по сравнению с их необобщенными аналогами.

- Обобщения обеспечивают лучшую производительность, т.к. лишены накладных расходов по упаковке/распаковке, когда хранят типы значений.
- Обобщения безопасны в отношении типов, потому что могут содержать только объекты указанного типа.
- Обобщения значительно сокращают потребность в специальных типах коллекций, поскольку при создании обобщенного контейнера указывается "вид типа".

Роль параметров обобщенных типов

Обобщенные классы, интерфейсы, структуры и делегаты вы можете обнаружить повсюду в библиотеках базовых классов .NET Core, и они могут быть частью любого пространства имен .NET Core. Кроме того, имейте в виду, что применение обобщений далеко не ограничивается простым определением класса коллекции. Разумеется, в оставшихся главах книги вы встретите случаи использования многих других обобщений для самых разных целей.

На заметку! Обобщенным образом могут быть записаны только классы, структуры, интерфейсы и делегаты, но не перечисления.

Глядя на обобщенный элемент в документации по .NET Core или в браузере объектов Visual Studio, вы заметите пару угловых скобок с буквой или другой лексемой внутри. На рис. 10.1 показано окно браузера объектов Visual Studio, в котором отображается набор обобщенных элементов из пространства имен `System.Collections.Generic`, включающий выделенный класс `List<T>`.

Формально эти лексемы называются *параметрами типа*, но в более дружественных к пользователю терминах на них можно ссылаться просто как на *заполнители*. Конструкцию `<T>` можно читать как "типа `T`". Таким образом, `IEnumerable<T>` можно прочитать как "`IEnumerable` типа `T`".

На заметку! Имя параметра типа (заполнитель) роли не играет и зависит от предпочтений разработчика, создавшего обобщенный элемент. Однако обычно имя `T` применяется для представления типов, `TKey` или `K` — для представления ключей и `TValue` или `V` — для представления значений.

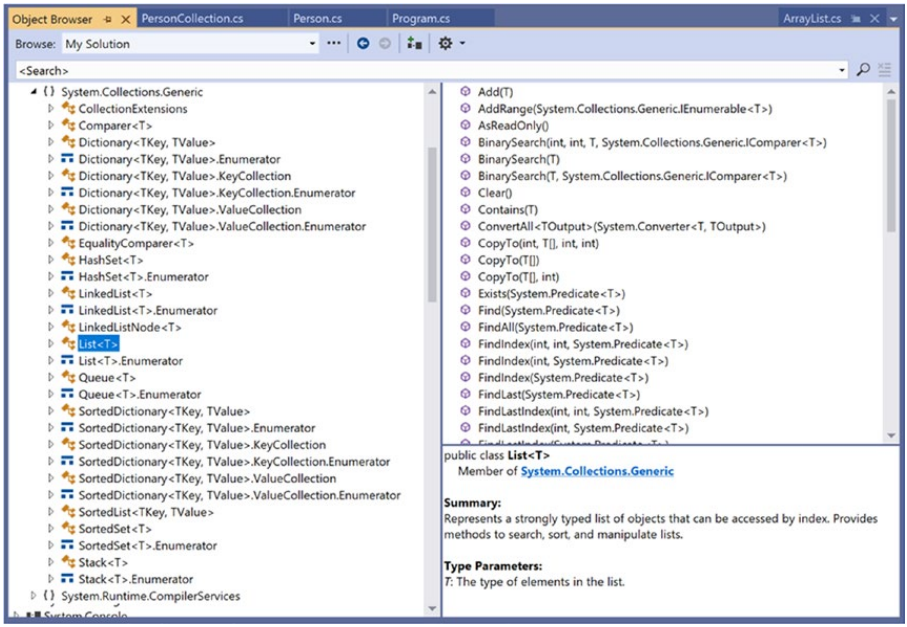

Рис. 10.1. Обобщенные элементы, поддерживающие параметры типа

Когда вы создаете обобщенный объект, реализуете обобщенный интерфейс или вызываете обобщенный член, на вас возлагается обязанность по предоставлению значения для параметра типа. Многочисленные примеры вы увидите как в этой главе, так и в остальных материалах книги. Тем не менее, для начала рассмотрим основы взаимодействия с обобщенными типами и членами.

Указание параметров типа для обобщенных классов и структур

При создании экземпляра обобщенного класса или структуры вы указываете параметр типа, когда объявляете переменную и когда вызываете конструктор. Как было показано в предыдущем фрагменте кода, в методе `UseGenericList()` определены два объекта `List<T>`:

```
// Этот объект List<> может хранить только объекты Person.
List<Person> morePeople = new List<Person>();

// Этот объект List<> может хранить только целые числа.
List<int> moreInts = new List<int>();
```

Первую строку приведенного выше кода можно трактовать как "список `List<>` объектов `T`, где `T` — тип `Person`" или более просто как "список объектов действующих лиц". После указания параметра типа обобщенного элемента изменить его нельзя (помните, что сущностью обобщений является безопасность в отношении типов). Когда параметр типа задается для обобщенного класса или структуры, все вхождения заполнителя (заполнителей) заменяются предоставленным значением.

Если вы просмотрите полное объявление обобщенного класса `List<T>` в браузере объектов Visual Studio, то заметите, что заполнитель `T` используется в определении повсеместно. Ниже приведен частичный листинг:

```
// Частичное определение класса List<T>.
namespace System.Collections.Generic
{
  public class List<T> : IList<T>, IList, IReadOnlyList<T>
  {
    ...
    public void Add(T item);
    public void AddRange(IEnumerable<T> collection);
    public ReadOnlyCollection<T> AsReadOnly();
    public int BinarySearch(T item);
    public bool Contains(T item);
    public void CopyTo(T[] array);
    public int FindIndex(System.Predicate<T> match);
    public T FindLast(System.Predicate<T> match);
    public bool Remove(T item);
    public int RemoveAll(System.Predicate<T> match);
    public T[] ToArray();
    public bool TrueForAll(System.Predicate<T> match);
    public T this[int index] { get; set; }
  }
}
```

В случае создания `List<T>` с указанием объектов `Person` результат будет таким же, как если бы тип `List<T>` был определен следующим образом:

```
namespace System.Collections.Generic
{
  public class List<Person> :
    IList<Person>, IList, IReadOnlyList<Person>
  {
    ...
    public void Add(Person item);
    public void AddRange(IEnumerable<Person> collection);
    public ReadOnlyCollection<Person> AsReadOnly();
    public int BinarySearch(Person item);
    public bool Contains(Person item);
    public void CopyTo(Person[] array);
    public int FindIndex(System.Predicate<Person> match);
    public Person FindLast(System.Predicate<Person> match);
    public bool Remove(Person item);
    public int RemoveAll(System.Predicate<Person> match);
    public Person[] ToArray();
    public bool TrueForAll(System.Predicate<Person> match);
    public Person this[int index] { get; set; }
  }
}
```

Несомненно, когда вы создаете в коде переменную обобщенного типа `List<T>`, компилятор вовсе не создает новую реализацию класса `List<T>`. Взамен он принимает во внимание только члены обобщенного типа, к которым вы действительно обращаетесь.

Указание параметров типа для обобщенных членов

В необобщенном классе или структуре разрешено поддерживать обобщенные свойства. В таких случаях необходимо также указывать значение заполнителя во время вызова метода. Например, класс `System.Array` поддерживает набор обобщенных методов. В частности, необобщенный статический метод `Sort()` имеет обобщенный аналог по имени `Sort<T>()`. Рассмотрим представленный далее фрагмент кода, где `T` — тип `int`:

```
int[] myInts = { 10, 4, 2, 33, 93 };
// Указание заполнителя для обобщенного метода Sort<>().
Array.Sort<int>(myInts);
foreach (int i in myInts)
{
  Console.WriteLine(i);
}
```

Указание параметров типов для обобщенных интерфейсов

Обобщенные интерфейсы обычно реализуются при построении классов или структур, которые нуждаются в поддержке разнообразных аспектов поведения платформы (скажем, клонирования, сортировки и перечисления). В главе 8 вы узнали о нескольких необобщенных интерфейсах, таких как `IComparable`, `IEnumerable`, `IEnumerator` и `IComparer`. Вспомните, что необобщенный интерфейс `IComparable` определен примерно так:

```
public interface IComparable
{
  int CompareTo(object obj);
}
```

В главе 8 этот интерфейс также был реализован классом `Car`, чтобы сделать возможной сортировку стандартного массива. Однако код требовал нескольких проверок времени выполнения и операций приведения, потому что параметром был общий тип `System.Object`:

```
public class Car : IComparable
{
  ...
  // Реализация IComparable.
  int IComparable.CompareTo(object obj)
  {
    if (obj is Car temp)
    {
      return this.CarID.CompareTo(temp.CarID);
    }
    throw new ArgumentException("Parameter is not a Car!");
                              // Параметр не является объектом типа Car!
  }
}
```

Теперь представим, что применяется обобщенный аналог данного интерфейса:

```
public interface IComparable<T>
{
  int CompareTo(T obj);
}
```

В таком случае код реализации будет значительно яснее:

```
public class Car : IComparable<Car>
{
  ...
  // Реализация IComparable<T>.
  int IComparable<Car>.CompareTo(Car obj)
  {
    if (this.CarID > obj.CarID)
    {
      return 1;
    }
    if (this.CarID < obj.CarID)
    {
      return -1;
    }
    return 0;
  }
}
```

Здесь уже не нужно проверять, относится ли входной параметр к типу `Car`, потому что он может быть *только* `Car`! В случае передачи несовместимого типа данных возникает ошибка на этапе компиляции. Теперь, углубив понимание того, как взаимодействовать с обобщенными элементами, а также усвоив роль параметров типа (т.е. заполнителей), вы готовы к исследованию классов и интерфейсов из пространства имен `System.Collections.Generic`.

Пространство имен `System.Collections.Generic`

Когда вы строите приложение .NET Core и необходим способ управления данными в памяти, классы из пространства имен `System.Collections.Generic` вероятно удовлетворят всем требованиям. В начале настоящей главы кратко упоминались некоторые основные необобщенные интерфейсы, реализуемые необобщенными классами коллекций. Не должен вызывать удивление тот факт, что в пространстве имен `System.Collections.Generic` для многих из них определены обобщенные замены.

В действительности вы сможете найти некоторое количество обобщенных интерфейсов, которые расширяют свои необобщенные аналоги, что может показаться странным. Тем не менее, за счет этого реализующие их классы будут также поддерживать унаследованную функциональность, которая имеется в их необобщенных родственных версиях. Например, интерфейс `IEnumerable<T>` расширяет `IEnumerable`. В табл. 10.4 описаны основные обобщенные интерфейсы, с которыми вы столкнетесь во время работы с обобщенными классами коллекций.

В пространстве имен `System.Collections.Generic` также определены классы, реализующие многие из указанных основных интерфейсов. В табл. 10.5 описаны часто используемые классы из этого пространства имен, реализуемые ими интерфейсы, а также их базовая функциональность.

Таблица 10.4. Основные интерфейсы, поддерживаемые классами из пространства имен `System.Collections.Generic`

Интерфейс `System.Collections.Generic`	Описание
`ICollection<T>`	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех типов обобщенных коллекций
`IComparer<T>`	Определяет способ сравнения объектов
`IDictionary<TKey, TValue>`	Позволяет объекту обобщенной коллекции представлять свое содержимое посредством пар "ключ-значение"
`IEnumerable<T>`/ `IAsyncEnumerable`	Возвращает интерфейс `IEnumerator<T>` для заданного объекта. Интерфейс `IAsyncEnumerable` (появившийся в версии C# 8.0) раскрывается в главе 15
`IEnumerator<T>`	Позволяет выполнять итерацию в стиле `foreach` по обобщенной коллекции
`IList<T>`	Обеспечивает поведение добавления, удаления и индексации элементов в последовательном списке объектов
`ISet<T>`	Предоставляет базовый интерфейс для абстракции множеств

Таблица 10.5. Классы из пространства имен `System.Collections.Generic`

Обобщенный класс	Поддерживаемые основные интерфейсы	Описание
`Dictionary<TKey, TValue>`	`ICollection<T>`, `IDictionary<TKey, TValue>`, `IEnumerable<T>`	Представляет обобщенную коллекцию ключей и значений
`LinkedList<T>`	`ICollection<T>`, `IEnumerable<T>`	Представляет двухсвязный список
`List<T>`	`ICollection<T>`, `IEnumerable<T>`, `IList<T>`	Представляет последовательный список элементов с динамически изменяемым размером
`Queue<T>`	`ICollection` (это не опечатка; именно так называется необобщенный интерфейс коллекции), `IEnumerable<T>`	Обобщенная реализация списка, работающего по алгоритму "первый вошел — первый вышел" (FIFO)
`SortedDictionary<TKey, TValue>`	`ICollection<T>`, `IDictionary<TKey, TValue>`, `IEnumerable<T>`	Обобщенная реализация сортированного множества пар "ключ-значение"
`SortedSet<T>`	`ICollection<T>`, `IEnumerable<T>`, `ISet<T>`	Представляет коллекцию объектов, поддерживаемых в сортированном порядке без дубликатов
`Stack<T>`	`ICollection` (это не опечатка; именно так называется необобщенный интерфейс коллекции), `IEnumerable<T>`	Обобщенная реализация списка, работающего по алгоритму "последний вошел — первый вышел" (LIFO)

В пространстве имен `System.Collections.Generic` также определены многие вспомогательные классы и структуры, которые работают в сочетании со специфическим контейнером. Например, тип `LinkedListNode<T>` представляет узел внутри обобщенного контейнера `LinkedList<T>`, исключение `KeyNotFoundException` генерируется при попытке получения элемента из коллекции с применением несуществующего ключа и т.д. Подробные сведения о пространстве имен `System.Collections.Generic` доступны в документации по .NET Core.

В любом случае следующая ваша задача состоит в том, чтобы научиться использовать некоторые из упомянутых классов обобщенных коллекций. Тем не менее, сначала полезно ознакомиться со средством языка C# (введенным в версии .NET 3.5), которое упрощает заполнение данными обобщенных (и необобщенных) коллекций.

Синтаксис инициализации коллекций

В главе 4 вы узнали о *синтаксисе инициализации массивов*, который позволяет устанавливать элементы новой переменной массива во время ее создания. С ним тесно связан *синтаксис инициализации коллекций*. Данное средство языка C# позволяет наполнять многие контейнеры (такие как `ArrayList` или `List<T>`) элементами с применением синтаксиса, похожего на тот, который используется для наполнения базовых массивов. Создайте новый проект консольного приложения .NET Core по имени `FunWithCollectionInitialization`. Удалите код, сгенерированный в `Program.cs`, и добавьте следующие операторы `using`:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Drawing;
```

На заметку! Синтаксис инициализации коллекций может применяться только к классам, которые поддерживают метод `Add()`, формально определяемый интерфейсами `ICollection<T>` и `ICollection`.

Взгляните на приведенные ниже примеры:

```
// Инициализация стандартного массива.
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализация обобщенного List<> с элементами int.
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализация ArrayList числовыми данными.
ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Если контейнером является коллекция классов или структур, тогда синтаксис инициализации коллекций можно смешивать с синтаксисом инициализации объектов, получая функциональный код. Вспомните класс `Point` из главы 5, в котором были определены два свойства, `X` и `Y`. Для построения обобщенного списка `List<T>` объектов `Point` можно написать такой код:

```
List<Point> myListOfPoints = new List<Point>
{
  new Point { X = 2, Y = 2 },
  new Point { X = 3, Y = 3 },
  new Point { X = 4, Y = 4 }
};
```

```
foreach (var pt in myListOfPoints)
{
  Console.WriteLine(pt);
}
```

Преимущество этого синтаксиса связано с сокращением объема клавиатурного ввода. Хотя вложенные фигурные скобки могут затруднять чтение кода, если не позаботиться о надлежащем форматировании, вы только вообразите себе объем кода, который пришлось бы написать для наполнения следующего списка `List<T>` объектов `Rectangle` без использования синтаксиса инициализации коллекций:

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
  new Rectangle {
    Height = 90, Width = 90,
    Location = new Point { X = 10, Y = 10 }},
  new Rectangle {
    Height = 50,Width = 50,
    Location = new Point { X = 2, Y = 2 }},
};
foreach (var r in myListOfRects)
{
  Console.WriteLine(r);
}
```

Работа с классом `List<T>`

Создайте новый проект консольного приложения под названием `FunWithGeneric Collections`. Добавьте новый файл по имени `Person.cs` и поместите в него показанный ниже код (это тот же самый код с определением предыдущего класса `Person`):

```
namespace FunWithGenericCollections
{
  public class Person
  {
    public int Age {get; set;}
    public string FirstName {get; set;}
    public string LastName {get; set;}

    public Person() {}
    public Person(string firstName, string lastName, int age)
    {
      Age = age;
      FirstName = firstName;
      LastName = lastName;
    }

    public override string ToString()
    {
      return $"Name: {FirstName} {LastName}, Age: {Age}";
    }
  }
}
```

Удалите сгенерированный код из файла `Program.cs` и добавьте следующие операторы `using`:

```
using System;
using System.Collections.Generic;
using FunWithGenericCollections;
```

Первым будет исследоваться обобщенный класс `List<T>`, который уже применялся ранее в главе. Класс `List<T>` используется чаще других классов из пространства имен `System.Collections.Generic`, т.к. он позволяет динамически изменять размер контейнера. Чтобы ознакомиться с его особенностями, добавьте в класс `Program` метод `UseGenericList()`, в котором задействован класс `List<T>` для манипулирования набором объектов `Person`; вспомните, что в классе `Person` определены три свойства (`Age`, `FirstName` и `LastName`), а также специальная реализация метода `ToString()`:

```
static void UseGenericList()
{
  // Создать список объектов Person и заполнить его с помощью
  // синтаксиса инициализации объектов и коллекции.
  List<Person> people = new List<Person>()
  {
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
  };

  // Вывести количество элементов в списке.
  Console.WriteLine("Items in list: {0}", people.Count);

  // Выполнить перечисление по списку.
  foreach (Person p in people)
    Console.WriteLine(p);

  // Вставить новый объект Person.
  Console.WriteLine("\n->Inserting new person.");
  people.Insert(2, new Person { FirstName = "Maggie",
                                LastName = "Simpson", Age = 2 });
  Console.WriteLine("Items in list: {0}", people.Count);

  // Скопировать данные в новый массив.
  Person[] arrayOfPeople = people.ToArray();
  foreach (Person p in arrayOfPeople)  // Вывести имена
  {
    Console.WriteLine("First Names: {0}", p.FirstName);
  }
}
```

Здесь для наполнения списка `List<T>` объектами применяется синтаксис инициализации в качестве сокращенной записи *многократного* вызова метода `Add()`. После вывода количества элементов в коллекции (и прохода по всем элементам) вызывается метод `Insert()`. Как видите, метод `Insert()` позволяет вставлять новый элемент в `List<T>` по указанному индексу.

Наконец, обратите внимание на вызов метода `ToArray()`, который возвращает массив объектов `Person`, основанный на содержимом исходного списка `List<T>`. Затем осуществляется проход по всем элементам данного массива с использованием синтаксиса индексатора массива. Вызов метода `UseGenericList()` в операторах верхнего уровня приводит к получению следующего вывода:

```
***** Fun with Generic Collections *****

Items in list: 4
Name: Homer Simpson, Age: 47
Name: Marge Simpson, Age: 45
Name: Lisa Simpson, Age: 9
Name: Bart Simpson, Age: 8

->Inserting new person.
Items in list: 5
First Names: Homer
First Names: Marge
First Names: Maggie
First Names: Lisa
First Names: Bart
```

В классе `List<T>` определено множество дополнительных членов, представляющих интерес, поэтому за полным их описанием обращайтесь в документацию. Давайте рассмотрим еще несколько обобщенных коллекций, в частности `Stack<T>`, `Queue<T>` и `SortedSet<T>`, что должно способствовать лучшему пониманию основных вариантов хранения данных в приложении.

Работа с классом `Stack<T>`

Класс `Stack<T>` представляет коллекцию элементов, которая обслуживает элементы в стиле "последний вошел — первый вышел" (LIFO). Как и можно было ожидать, в `Stack<T>` определены члены `Push()` и `Pop()`, предназначенные для вставки и удаления элементов из стека. Приведенный ниже метод создает стек объектов `Person`:

```
static void UseGenericStack()
{
  Stack<Person> stackOfPeople = new();
  stackOfPeople.Push(new Person { FirstName = "Homer",
                                  LastName = "Simpson", Age = 47 });
  stackOfPeople.Push(new Person { FirstName = "Marge",
                                  LastName = "Simpson", Age = 45 });
  stackOfPeople.Push(new Person { FirstName = "Lisa",
                                  LastName = "Simpson", Age = 9 });
  // Просмотреть верхний элемент, вытолкнуть его и просмотреть снова.
  Console.WriteLine("First person is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
  Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
  Console.WriteLine("\nFirst person item is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

  try
  {
    Console.WriteLine("\nnFirst person is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine("\nError! {0}", ex.Message);  // Ошибка! Стек пуст.
  }
}
```

В коде строится стек, который содержит информацию о трех лицах, добавленных в алфавитном порядке следования их имен: Homer, Marge и Lisa. Заглядывая (посредством метода Peek()) в стек, вы будете всегда видеть объект, находящийся на его вершине; следовательно, первый вызов Peek() возвращает третий объект Person. После серии вызовов Pop() и Peek() стек, в конце концов, опустошается, после чего дополнительные вызовы Peek() и Pop() приводят к генерации системного исключения. Вот как выглядит вывод:

```
***** Fun with Generic Collections *****

First person is: Name: Lisa Simpson, Age: 9
Popped off Name: Lisa Simpson, Age: 9

First person is: Name: Marge Simpson, Age: 45
Popped off Name: Marge Simpson, Age: 45

First person item is: Name: Homer Simpson, Age: 47
Popped off Name: Homer Simpson, Age: 47

Error! Stack empty.
```

Работа с классом Queue<T>

Очереди — это контейнеры, которые обеспечивают доступ к элементам в стиле "первый вошел — первый вышел" (FIFO). К сожалению, людям приходится сталкиваться с очередями практически ежедневно: в банке, в супермаркете, в кафе. Когда нужно смоделировать сценарий, в котором элементы обрабатываются в режиме FIFO, класс Queue<T> подходит наилучшим образом. Дополнительно к функциональности, предоставляемой поддерживаемыми интерфейсами, в Queue определены основные члены, перечисленные в табл. 10.6.

Таблица 10.6. Члены типа Queue<T>

Член Queue<T>	Описание
Dequeue()	Удаляет и возвращает объект из начала Queue<T>
Enqueue()	Добавляет объект в конец Queue<T>
Peek()	Возвращает объект из начала Queue<T>, не удаляя его

Теперь давайте посмотрим на описанные методы в работе. Можно снова задействовать класс Person и построить объект Queue<T>, эмулирующий очередь людей, которые ожидают заказанный кофе.

```
static void UseGenericQueue()
{
  // Создать очередь из трех человек.
  Queue<Person> peopleQ = new();
  peopleQ.Enqueue(new Person {FirstName= "Homer",
                              LastName="Simpson", Age=47});
  peopleQ.Enqueue(new Person {FirstName= "Marge",
                              LastName="Simpson", Age=45});
  peopleQ.Enqueue(new Person {FirstName= "Lisa",
                              LastName="Simpson", Age=9});

  // Заглянуть, кто первый в очереди.
  Console.WriteLine("{0} is first in line!", peopleQ.Peek().FirstName);
```

```
    // Удалить всех из очереди.
    GetCoffee(peopleQ.Dequeue());
    GetCoffee(peopleQ.Dequeue());
    GetCoffee(peopleQ.Dequeue());

    // Попробовать извлечь кого-то из очереди снова.
    try
    {
      GetCoffee(peopleQ.Dequeue());
    }
    catch(InvalidOperationException e)
    {
      Console.WriteLine("Error! {0}", e.Message); //Ошибка! Очередь пуста.
    }

    // Локальная вспомогательная функция.
    static void GetCoffee(Person p)
    {
      Console.WriteLine("{0} got coffee!", p.FirstName);
    }
  }
```

Здесь с применением метода `Enqueue()` в `Queue<T>` вставляются три элемента. Вызов `Peek()` позволяет просматривать (но не удалять) первый элемент, находящийся в текущий момент внутри `Queue`. Наконец, вызов `Dequeue()` удаляет элемент из очереди и передает его на обработку вспомогательной функции `GetCoffee()`. Обратите внимание, что если попытаться удалить элемент из пустой очереди, то сгенерируется исключение времени выполнения. Ниже показан вывод, полученный в результате вызова метода `UseGenericQueue()`:

```
***** Fun with Generic Collections *****

Homer is first in line!
Homer got coffee!
Marge got coffee!
Lisa got coffee!
Error! Queue empty.
```

Работа с классом `SortedSet<T>`

Класс `SortedSet<T>` полезен тем, что при вставке или удалении элементов он автоматически обеспечивает сортировку элементов в наборе. Однако классу `SortedSet<T>` необходимо сообщить, *каким образом* должны сортироваться объекты, путем передачи его конструктору в качестве аргумента объекта, который реализует обобщенный интерфейс `IComparer<T>`.

Начните с создания нового класса по имени `SortPeopleByAge`, реализующего интерфейс `IComparer<T>`, где `T` — тип `Person`. Вспомните, что в этом интерфейсе определен единственный метод по имени `Compare()`, в котором можно запрограммировать логику сравнения элементов. Вот простая реализация:

```
using System.Collections.Generic;

namespace FunWithGenericCollections
{
  class SortPeopleByAge : IComparer<Person>
  {
```

```
    public int Compare(Person firstPerson, Person secondPerson)
    {
      if (firstPerson?.Age > secondPerson?.Age)
      {
        return 1;
      }
      if (firstPerson?.Age < secondPerson?.Age)
      {
        return -1;
      }
      return 0;
    }
  }
}
```

Теперь добавьте в класс `Program` следующий новый метод, который позволит продемонстрировать применение `SortedSet<Person>`:

```
static void UseSortedSet()
{
  // Создать несколько объектов Person с разными значениями возраста.
  SortedSet<Person> setOfPeople = new SortedSet<Person>(new SortPeopleByAge())
  {
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
  };
  // Обратите внимание, что элементы отсортированы по возрасту.
  foreach (Person p in setOfPeople)
  {
    Console.WriteLine(p);
  }
    Console.WriteLine();
  // Добавить еще несколько объектов Person с разными значениями возраста.
  setOfPeople.Add(new Person { FirstName = "Saku",
                               LastName = "Jones", Age = 1 });
  setOfPeople.Add(new Person { FirstName = "Mikko",
                               LastName = "Jones", Age = 32 });
  // Элементы по-прежнему отсортированы по возрасту.
  foreach (Person p in setOfPeople)
  {
    Console.WriteLine(p);
  }
}
```

Запустив приложение, легко заметить, что список объектов будет всегда упорядочен на основе значения свойства `Age` независимо от порядка вставки и удаления объектов:

```
***** Fun with Generic Collections *****

Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47
```

```
Name: Saku Jones, Age: 1
Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Mikko Jones, Age: 32
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47
```

Работа с классом Dictionary<TKey, TValue>

Еще одной удобной обобщенной коллекцией является класс Dictionary<TKey, TValue>, позволяющий хранить любое количество объектов, на которые можно ссылаться через уникальный ключ. Таким образом, вместо получения элемента из List<T> с использованием числового идентификатора (например, "извлечь второй объект") можно применять уникальный строковый ключ (скажем, "предоставить объект с ключом Homer").

Как и другие классы коллекций, наполнять Dictionary<TKey,TValue> можно путем вызова обобщенного метода Add() вручную. Тем не менее, заполнять Dictionary<TKey,TValue> допускается также с использованием синтаксиса инициализации коллекций. Имейте в виду, что при наполнении данного объекта коллекции ключи должны быть уникальными. Если вы по ошибке укажете один и тот же ключ несколько раз, то получите исключение времени выполнения.

Взгляните на следующий метод, который наполняет Dictionary<K,V> разнообразными объектами. Обратите внимание, что при создании объекта Dictionary<TKey,TValue> в качестве аргументов конструктора передаются тип ключа (TKey) и тип внутренних объектов (TValue). В этом примере для ключа указывается тип данных string, а для значения — тип Person. Кроме того, имейте в виду, что синтаксис инициализации объектов можно сочетать с синтаксисом инициализации коллекций.

```
private static void UseDictionary()
{
  // Наполнить с помощью метода Add().
  Dictionary<string, Person> peopleA = new Dictionary<string, Person>();
  peopleA.Add("Homer", new Person { FirstName = "Homer",
                                    LastName = "Simpson", Age = 47 });
  peopleA.Add("Marge", new Person { FirstName = "Marge",
                                    LastName = "Simpson", Age = 45 });
  peopleA.Add("Lisa", new Person { FirstName = "Lisa",
                                   LastName = "Simpson", Age = 9 });

  // Получить элемент с ключом Homer.
  Person homer = peopleA["Homer"];
  Console.WriteLine(homer);

  // Наполнить с помощью синтаксиса инициализации.
  Dictionary<string, Person> peopleB = new Dictionary<string, Person>()
  {
    { "Homer", new Person { FirstName = "Homer",
                            LastName = "Simpson", Age = 47 } },
    { "Marge", new Person { FirstName = "Marge",
                            LastName = "Simpson", Age = 45 } },
    { "Lisa", new Person { FirstName = "Lisa",
                           LastName = "Simpson", Age = 9 } }
  };
```

```
  // Получить элемент с ключом Lisa.
  Person lisa = peopleB["Lisa"];
  Console.WriteLine(lisa);
}
```

Наполнять `Dictionary<TKey,TValue>` также возможно с применением связанного синтаксиса инициализации, который является специфичным для контейнера данного типа (вполне ожидаемо называемый *инициализацией словарей*). Подобно синтаксису, который использовался при наполнении объекта `personB` в предыдущем примере, для объекта коллекции определяется область инициализации; однако можно также применять индексатор, чтобы указать ключ, и присвоить ему новый объект:

```
// Наполнить с помощью синтаксиса инициализации словарей.
Dictionary<string, Person> peopleC = new Dictionary<string, Person>()
{
  ["Homer"] = new Person { FirstName = "Homer",
                           LastName = "Simpson", Age = 47 },
  ["Marge"] = new Person { FirstName = "Marge",
                           LastName = "Simpson", Age = 45 },
  ["Lisa"] = new Person { FirstName = "Lisa",
                          LastName = "Simpson", Age = 9 }
};
```

Пространство имен System.Collections.ObjectModel

Теперь, когда вы понимаете, как работать с основными обобщенными классами, можно кратко рассмотреть дополнительное пространство имен, связанное с коллекциями — `System.Collections.ObjectModel`. Это относительно небольшое пространство имен, содержащее совсем мало классов. В табл. 10.7 документированы два класса, о которых вы обязательно должны быть осведомлены.

Таблица 10.7. Полезные классы из пространства имен System.Collections.ObjectModel

Класс System.Collections.ObjectModel	Описание
`ObservableCollection<T>`	Представляет динамическую коллекцию данных, которая обеспечивает уведомление при добавлении и удалении элементов, а также при обновлении всего списка
`ReadOnlyObservableCollection<T>`	Представляет версию `ObservableCollection<T>`, допускающую только чтение

Класс `ObservableCollection<T>` удобен своей возможностью информировать внешние объекты, когда его содержимое каким-то образом изменяется (как и можно было догадаться, работа с `ReadOnlyObservableCollection<T>` похожа, но по своей природе допускает только чтение).

Работа с классом ObservableCollection<T>

Создайте новый проект консольного приложения по имени FunWithObservable Collections и импортируйте в первоначальный файл кода C# пространство имен System.Collections.ObjectModel. Во многих отношениях работа с ObservableCollection<T> идентична работе с List<T>, учитывая, что оба класса реализуют те же самые основные интерфейсы. Уникальным класс ObservableCollection<T> делает тот факт, что он поддерживает событие по имени CollectionChanged. Указанное событие будет инициироваться каждый раз, когда вставляется новый элемент, удаляется (или перемещается) существующий элемент либо модифицируется вся коллекция целиком.

Подобно любому другому событию событие CollectionChanged определено в терминах делегата, которым в данном случае является NotifyCollectionChangedEven tHandler. Этот делегат может вызывать любой метод, который принимает object в первом параметре и NotifyCollectionChangedEventArgs — во втором. Рассмотрим следующий код, в котором наполняется наблюдаемая коллекция, содержащая объекты Person, и осуществляется привязка к событию CollectionChanged:

```
using System;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using FunWithObservableCollections;

// Сделать коллекцию наблюдаемой
// и добавить в нее несколько объектов Person.
ObservableCollection<Person> people =
     new ObservableCollection<Person>()
{
  new Person{ FirstName = "Peter", LastName = "Murphy", Age = 52 },
  new Person{ FirstName = "Kevin", LastName = "Key", Age = 48 },
};

// Привязаться к событию CollectionChanged.
people.CollectionChanged += people_CollectionChanged;
static void people_CollectionChanged(object sender,
   System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
  throw new NotImplementedException();
}
```

Входной параметр NotifyCollectionChangedEventArgs определяет два важных свойства, OldItems и NewItems, которые выдают список элементов, имеющихся в коллекции перед генерацией события, и список новых элементов, вовлеченных в изменение. Тем не менее, такие списки будут исследоваться только в подходящих обстоятельствах. Вспомните, что событие CollectionChanged инициируется при добавлении, удалении, перемещении или сбросе элементов. Чтобы выяснить, какое из упомянутых действий запустило событие, можно использовать свойство Action объекта NotifyC ollectionChangedEventArgs. Свойство Action допускается проверять на предмет равенства любому из членов перечисления NotifyCollectionChangedAction:

```
public enum NotifyCollectionChangedAction
{
  Add = 0,
  Remove = 1,
```

```
  Replace = 2,
  Move = 3,
  Reset = 4,
}
```

Ниже показана реализация обработчика событий `CollectionChanged`, который будет обходить старый и новый наборы, когда элемент вставляется или удаляется из имеющейся коллекции (обратите внимание на оператор `using` для `System.Collections.Specialized`):

```
using System.Collections.Specialized;
...
static void people_CollectionChanged(object sender,
   NotifyCollectionChangedEventArgs e)
{
  // Выяснить действие, которое привело к генерации события.
  Console.WriteLine("Action for this event: {0}", e.Action);

  // Было что-то удалено.
  if (e.Action == NotifyCollectionChangedAction.Remove)
  {
    Console.WriteLine("Here are the OLD items:");  // старые элементы
    foreach (Person p in e.OldItems)
    {
      Console.WriteLine(p.ToString());
    }
    Console.WriteLine();
  }

  // Было что-то добавлено.
  if (e.Action == NotifyCollectionChangedAction.Add)
  {
    // Теперь вывести новые элементы, которые были вставлены.
    Console.WriteLine("Here are the NEW items:");  // Новые элементы
    foreach (Person p in e.NewItems)
    {
      Console.WriteLine(p.ToString());
    }
  }
}
```

Модифицируйте вызывающий код для добавления и удаления элемента:

```
// Добавить новый элемент.
people.Add(new Person("Fred", "Smith", 32));

// Удалить элемент.
people.RemoveAt(0);
```

В результате запуска программы вы получите вывод следующего вида:

```
Action for this event: Add
Here are the NEW items:
Name: Fred Smith, Age: 32

Action for this event: Remove
Here are the OLD items:
Name: Peter Murphy, Age: 52
```

На этом исследование различных пространств имен, связанных с коллекциями, завершено. В конце главы будет также объясняться, как и для чего строить собственные обобщенные методы и обобщенные типы.

Создание специальных обобщенных методов

Несмотря на то что большинство разработчиков обычно применяют обобщенные типы, имеющиеся в библиотеках базовых классов, существует также возможность построения собственных обобщенных методов и специальных обобщенных типов. Давайте посмотрим, как включать обобщения в свои проекты. Первым делом будет построен обобщенный метод обмена. Начните с создания нового проекта консольного приложения по имени `CustomGenericMethods`.

Построение специальных обобщенных методов представляет собой более развитую версию традиционной перегрузки методов. В главе 2 вы узнали, что перегрузка — это действие по определению нескольких версий одного метода, которые отличаются друг от друга количеством или типами параметров.

Хотя перегрузка является полезным средством объектно-ориентированного языка, проблема заключается в том, что при этом довольно легко получить в итоге огромное количество методов, которые по существу делают одно и то же. Например, пусть необходимо создать методы, которые позволяют менять местами два фрагмента данных посредством простой процедуры. Вы можете начать с написания нового статического класса с методом, который способен оперировать целочисленными значениями:

```
using System;
namespace CustomGenericMethods
{
  static class SwapFunctions
  {
    // Поменять местами два целочисленных значения.
    static void Swap(ref int a, ref int b)
    {
      int temp = a;
      a = b;
      b = temp;
    }
  }
}
```

Пока все идет хорошо. Но теперь предположим, что нужно менять местами также и два объекта `Person`; действие потребует написания новой версии метода `Swap()`:

```
// Поменять местами два объекта Person.
static void Swap(ref Person a, ref Person b)
{
  Person temp = a;
  a = b;
  b = temp;
}
```

Вне всяких сомнений вам должно быть ясно, чем все закончится. Если также понадобится менять местами два значения с плавающей точкой, два объекта растровых изображений, два объекта автомобилей, два объекта кнопок или что-нибудь еще, то придется писать дополнительные методы, что в итоге превратится в настоящий кошмар при сопровождении. Можно было бы построить один (необобщенный) метод,

оперирующий с параметрами типа object, но тогда возвратятся все проблемы, которые были описаны ранее в главе, т.е. упаковка, распаковка, отсутствие безопасности в отношении типов, явное приведение и т.д.

Наличие группы перегруженных методов, отличающихся только входными аргументами — явный признак того, что обобщения могут облегчить ситуацию. Рассмотрим следующий обобщенный метод Swap<T>(), который способен менять местами два значения типа T:

```
// Этот метод будет менять местами два элемента
// типа, указанного в параметре <T>.
static void Swap<T>(ref T a, ref T b)
{
  Console.WriteLine(«You sent the Swap() method a {0}», typeof(T));
  T temp = a;
  a = b;
  b = temp;
}
```

Обратите внимание, что обобщенный метод определен за счет указания параметра типа после имени метода, но перед списком параметров. Здесь заявлено, что метод Swap<T>() способен оперировать на любых двух параметрах типа <T>. Для придания некоторой пикантности имя замещаемого типа выводится на консоль с использованием операции typeof() языка C#. Взгляните на показанный ниже вызывающий код, который меняет местами целочисленные и строковые значения:

```
Console.WriteLine("***** Fun with Custom Generic Methods *****\n");

// Поменять местами два целочисленных значения.
int a = 10, b = 90;
Console.WriteLine("Before swap: {0}, {1}", a, b);
SwapFunctions.Swap<int>(ref a, ref b);
Console.WriteLine("After swap: {0}, {1}", a, b);
Console.WriteLine();

// Поменять местами два строковых значения.
string s1 = "Hello", s2 = "There";
Console.WriteLine("Before swap: {0} {1}!", s1, s2);
SwapFunctions.Swap<string>(ref s1, ref s2);
Console.WriteLine("After swap: {0} {1}!", s1, s2);

Console.ReadLine();
```

Вот вывод:

```
***** Fun with Custom Generic Methods *****

Before swap: 10, 90
You sent the Swap() method a System.Int32
After swap: 90, 10

Before swap: Hello There!
You sent the Swap() method a System.String
After swap: There Hello!
```

Главное преимущество такого подхода в том, что придется сопровождать только одну версию Swap<T>(), однако она в состоянии работать с любыми двумя элементами заданного типа в безопасной в отношении типов манере. Еще лучше то, что находящиеся в стеке элементы остаются в стеке, а расположенные в куче — соответственно в куче.

Выведение параметров типа

При вызове обобщенных методов вроде `Swap<T>()` параметр типа можно опускать, если (и только если) обобщенный метод принимает аргументы, поскольку компилятор в состоянии вывести параметр типа на основе параметров членов. Например, добавив к операторам верхнего уровня следующий код, можно менять местами два значения `System.Boolean`:

```
// Компилятор выведет тип System.Boolean.
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
SwapFunctions.Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

Несмотря на то что компилятор может определить параметр типа на основе типа данных, который применялся в объявлениях `b1` и `b2`, вы должны выработать привычку всегда указывать параметр типа явно:

```
SwapFunctions.Swap<bool>(ref b1, ref b2);
```

Такой подход позволяет другим программистам понять, что метод на самом деле является обобщенным. Кроме того, выведение типов параметров работает только в случае, если обобщенный метод принимает, по крайней мере, один параметр. Например, пусть в классе `Program` определен обобщенный метод `DisplayBaseClass<T>()`:

```
static void DisplayBaseClass<T>()
{
  // BaseType - метод, используемый в рефлексии;
  // он будет описан в главе 17.
  Console.WriteLine("Base class of {0} is: {1}.",
                    typeof(T), typeof(T).BaseType);
}
```

В таком случае при его вызове потребуется указать параметр типа:

```
...
// Если метод не принимает параметров,
// то должен быть указан параметр типа.
DisplayBaseClass<int>();
DisplayBaseClass<string>();

// Ошибка на этапе компиляции! Нет параметров?
// Должен быть предоставлен заполнитель!
// DisplayBaseClass();
Console.ReadLine();
}
```

Разумеется, обобщенные методы не обязаны быть статическими, как в приведенных выше примерах. Кроме того, применимы все правила и варианты для необобщенных методов.

Создание специальных обобщенных структур и классов

Так как вы уже знаете, каким образом определять и вызывать обобщенные методы, наступило время уделить внимание конструированию обобщенной структуры

(процесс построения обобщенного класса идентичен) в новом проекте консольного приложения по имени GenericPoint. Предположим, что вы построили обобщенную структуру Point, которая поддерживает единственный параметр типа, определяющий внутреннее представление координат (x, y). Затем в вызывающем коде можно создавать типы Point<T>:

```
// Точка с координатами типа int.
Point<int> p = new Point<int>(10, 10);

// Точка с координатами типа double.
Point<double> p2 = new Point<double>(5.4, 3.3);

// Точка с координатами типа string.
Point<string> p3 = new Point<string>("","3");
```

Создание точки с использованием строк поначалу может показаться несколько странным, но возьмем случай мнимых чисел, и тогда применение строк для значений X и Y точки может обрести смысл. Так или иначе, такая возможность демонстрирует всю мощь обобщений. Вот полное определение структуры Point<T>:

```
namespace GenericPoint
{
  // Обобщенная структура Point.
  public struct Point<T>
  {
    // Обобщенные данные состояния.
    private T _xPos;
    private T _yPos;

    // Обобщенный конструктор.
    public Point(T xVal, T yVal)
    {
      _xPos = xVal;
      _yPos = yVal;
    }

    // Обобщенные свойства.
    public T X
    {
      get => _xPos;
      set => _xPos = value;
    }

    public T Y
    {
      get => _yPos;
      set => _yPos = value;
    }

    public override string ToString() => $"[{_xPos}, {_yPos}]";
  }
}
```

Как видите, структура Point<T> задействует параметр типа в определениях полей данных, в аргументах конструктора и в определениях свойств.

Выражения default вида значений в обобщениях

С появлением обобщений ключевое слово default получило двойную идентичность. Вдобавок к использованию внутри конструкции switch оно также может применяться для установки параметра типа в стандартное значение. Это очень удобно, т.к. действительные типы, подставляемые вместо заполнителей, обобщенному типу заранее не известны, а потому он не может безопасно предполагать, какими будут стандартные значения. Параметры типа подчиняются следующим правилам:

- числовые типы имеют стандартное значение 0;
- ссылочные типы имеют стандартное значение null;
- поля структур устанавливаются в 0 (для типов значений) или в null (для ссылочных типов).

Чтобы сбросить экземпляр Point<T> в начальное состояние, значения X и Y можно было бы установить в 0 напрямую. Это предполагает, что вызывающий код будет предоставлять только числовые данные. А как насчет версии string? Именно здесь пригодится синтаксис default(T). Ключевое слово default сбрасывает переменную в стандартное значение для ее типа данных. Добавьте метод по имени ResetPoint():

```
// Сбросить поля в стандартное значение параметра типа.
// Ключевое слово default в языке C# перегружено.
// При использовании с обобщениями оно представляет
// стандартное значение параметра типа.
public void ResetPoint()
{
  _xPos = default(T);
  _yPos = default(T);
}
```

Теперь, располагая методом ResetPoint(), вы можете в полной мере использовать методы структуры Point<T>.

```
using System;
using GenericPoint;
Console.WriteLine("***** Fun with Generic Structures *****\n");
// Точка с координатами типа int.
Point<int> p = new Point<int>(10, 10);
Console.WriteLine("p.ToString()={0}", p.ToString());
p.ResetPoint();
Console.WriteLine("p.ToString()={0}", p.ToString());
Console.WriteLine();
// Точка с координатами типа double.
Point<double> p2 = new Point<double>(5.4, 3.3);
Console.WriteLine("p2.ToString()={0}", p2.ToString());
p2.ResetPoint();
Console.WriteLine("p2.ToString()={0}", p2.ToString());
Console.WriteLine();
// Точка с координатами типа string.
Point<string> p3 = new Point<string>("i", "3i");
Console.WriteLine("p3.ToString()={0}", p3.ToString());
p3.ResetPoint();
Console.WriteLine("p3.ToString()={0}", p3.ToString());
Console.ReadLine();
```

Ниже приведен вывод:

```
***** Fun with Generic Structures *****

p.ToString()=[10, 10]
p.ToString()=[0, 0]

p2.ToString()=[5.4, 3.3]
p2.ToString()=[0, 0]

p3.ToString()=[i, 3i]
p3.ToString()=[, ]
```

Выражения `default` литерального вида (нововведение в версии 7.1)

В дополнение к установке стандартного значения свойства в версии C# 7.1 появились выражения `default` литерального вида, которые устраняют необходимость в указании типа переменной в `default`. Модифицируйте метод `ResetPoint()`, как показано ниже:

```
public void ResetPoint()
{
  _xPos = default;
  _yPos = default;
}
```

Выражение `default` не ограничивается простыми переменными и может также применяться к сложным типам. Например, вот как можно создать и инициализировать структуру `Point`:

```
Point<string> p4 = default;
Console.WriteLine("p4.ToString()={0}", p4.ToString());
Console.WriteLine();
Point<int> p5 = default;
Console.WriteLine("p5.ToString()={0}", p5.ToString());
```

Сопоставление с образцом в обобщениях (нововведение в версии 7.1)

Еще одним обновлением в версии C# 7.1 является возможность использования сопоставления с образцом в обобщениях. Взгляните на приведенный далее метод, проверяющий экземпляр `Point` на предмет типа данных, на котором он основан (вероятно, неполный, но достаточный для того, чтобы продемонстрировать концепцию):

```
static void PatternMatching<T>(Point<T> p)
{
  switch (p)
  {
    case Point<string> pString:
      Console.WriteLine("Point is based on strings");
                    // Структура Point основана на типе string
      return;
    case Point<int> pInt:
      Console.WriteLine("Point is based on ints");
                    // Структура Point основана на типе int
      return;
  }
}
```

Для использования кода сопоставления с образцом модифицируйте операторы верхнего уровня следующим образом:

```
Point<string> p4 = default;
Point<int> p5 = default;
PatternMatching(p4);
PatternMatching(p5);
```

Ограничение параметров типа

Как объяснялось в настоящей главе, любой обобщенный элемент имеет, по крайней мере, один параметр типа, который необходимо указывать во время взаимодействия с данным обобщенным типом или его членом. Уже одно это обстоятельство позволяет строить код, безопасный в отношении типов; тем не менее, вы также можете применять ключевое слово `where` для определения особых требований к отдельному параметру типа.

С помощью ключевого слова `where` можно добавлять набор ограничений к конкретному параметру типа, которые компилятор C# проверит на этапе компиляции. В частности, параметр типа можно ограничить, как описано в табл. 10.8.

Таблица 10.8. Возможные ограничения для параметров типа в обобщениях

Ограничение	Описание
`where T : struct`	Параметр типа <T> должен иметь в своей цепочке наследования класс `System.ValueType` (т.е. <T> должен быть структурой)
`where T : class`	Параметр типа <T> *не* должен иметь в своей цепочке наследования класс `System.ValueType` (т.е. <T> должен быть ссылочным типом)
`where T : new()`	Параметр типа <T> обязан иметь стандартный конструктор. Это полезно, если обобщенный тип должен создавать экземпляры параметра типа, т.к. предугадать формат специальных конструкторов невозможно. Обратите внимание, что в типе с множеством ограничений данное ограничение должно указываться последним
`where T :` *ИмяБазовогоКласса*	Параметр типа <T> должен быть производным от класса, указанного как *ИмяБазовогоКласса*
`where T :` *ИмяИнтерфейса*	Параметр типа <T> должен реализовывать интерфейс, указанный как *ИмяИнтерфейса*. Можно задавать список из нескольких интерфейсов, разделяя их запятыми

Возможно, применять ключевое слово `where` в проектах C# вам никогда и не придется, если только не требуется строить какие-то исключительно безопасные в отношении типов специальные коллекции. Невзирая на сказанное, в следующих нескольких примерах (частичного) кода демонстрируется работа с ключевым словом `where`.

Примеры использования ключевого слова `where`

Начнем с предположения о том, что создан специальный обобщенный класс, и необходимо гарантировать наличие в параметре типа стандартного конструктора. Это может быть полезно, когда специальный обобщенный класс должен создавать экземпляры типа `T`, потому что стандартный конструктор является единственным конс-

труктором, потенциально общим для всех типов. Кроме того, подобное ограничение T позволяет получить проверку на этапе компиляции; если T — ссылочный тип, то программист будет помнить о повторном определении стандартного конструктора в объявлении класса (как вам уже известно, в случае определения собственного конструктора класса стандартный конструктор из него удаляется).

```
// Класс MyGenericClass является производным от object, в то время как
// содержащиеся в нем элементы должны иметь стандартный конструктор.
public class MyGenericClass<T> where T : new()
{
  ...
}
```

Обратите внимание, что конструкция where указывает параметр типа, подлежащий ограничению, за которым следует операция двоеточия. После операции двоеточия перечисляются все возможные ограничения (в данном случае — стандартный конструктор). Вот еще один пример:

```
// Класс MyGenericClass является производным от object, в то время как
// содержащиеся в нем элементы должны относиться к классу, реализующему
// интерфейс IDrawable, и поддерживать стандартный конструктор.
public class MyGenericClass<T> where T : class, IDrawable, new()
{
  ...
}
```

Здесь к типу T предъявляются три требования. Во-первых, он должен быть ссылочным типом (не структурой), как помечено лексемой class. Во-вторых, T должен реализовывать интерфейс IDrawable. В-третьих, тип T также должен иметь стандартный конструктор. Множество ограничений перечисляются в виде списка с разделителями-запятыми, но имейте в виду, что ограничение new() должно указываться последним! Таким образом, представленный далее код не скомпилируется:

```
// Ошибка! Ограничение new() должно быть последним в списке!
public class MyGenericClass<T> where T : new(), class, IDrawable
{
  ...
}
```

При создании класса обобщенной коллекции с несколькими параметрами типа можно указывать уникальный набор ограничений для каждого параметра, применяя отдельные конструкции where:

```
// Тип <K> должен расширять SomeBaseClass и иметь стандартный конструктор,
// в то время как тип <T> должен быть структурой и реализовывать
// обобщенный интерфейс IComparable.
public class MyGenericClass<K, T> where K : SomeBaseClass, new()
  where T : struct, IComparable<T>
{
  ...
}
```

Необходимость построения полного специального обобщенного класса коллекции возникает редко; однако ключевое слово where допускается использовать также в обобщенных методах. Например, если нужно гарантировать, что метод Swap<T>() может работать только со структурами, измените его код следующим образом:

```
// Этот метод меняет местами любые структуры, но не классы.
static void Swap<T>(ref T a, ref T b) where T : struct
{
  ...
}
```

Обратите внимание, что если ограничить метод `Swap<T>()` в подобной манере, то менять местами объекты `string` (как было показано в коде примера) больше не удастся, т.к. `string` является ссылочным типом.

Отсутствие ограничений операций

В завершение главы следует упомянуть об еще одном факте, связанном с обобщенными методами и ограничениями. При создании обобщенных методов может оказаться неожиданным получение ошибки на этапе компиляции в случае применения к параметрам типа любых операций C# (+, -, *, == и т.д.). Например, только вообразите, насколько полезным оказался бы класс, способный выполнять сложение, вычитание, умножение и деление с обобщенными типами:

```
// Ошибка на этапе компиляции! Невозможно
// применять операции к параметрам типа!
public class BasicMath<T>
{
  public T Add(T arg1, T arg2)
  { return arg1 + arg2; }
  public T Subtract(T arg1, T arg2)
  { return arg1 - arg2; }
  public T Multiply(T arg1, T arg2)
  { return arg1 * arg2; }
  public T Divide(T arg1, T arg2)
  { return arg1 / arg2; }
}
```

К сожалению, приведенный выше класс `BasicMath<T>` не скомпилируется. Хотя это может показаться крупным недостатком, следует вспомнить, что обобщения имеют общий характер. Конечно, числовые данные прекрасно работают с двоичными операциями C#. Тем не менее, справедливости ради, если аргумент `<T>` является специальным классом или структурой, то компилятор мог бы предположить, что он поддерживает операции +, -, * и /. В идеале язык C# позволял бы ограничивать обобщенный тип поддерживаемыми операциями, как показано ниже:

```
// Только в целях иллюстрации!
public class BasicMath<T> where T : operator +, operator -,
  operator *, operator /
{
  public T Add(T arg1, T arg2)
  { return arg1 + arg2; }
  public T Subtract(T arg1, T arg2)
  { return arg1 - arg2; }
  public T Multiply(T arg1, T arg2)
  { return arg1 * arg2; }
  public T Divide(T arg1, T arg2)
  { return arg1 / arg2; }
}
```

Увы, ограничения операций в текущей версии C# не поддерживаются. Однако достичь желаемого результата можно (хотя и с дополнительными усилиями) путем определения интерфейса, который поддерживает такие операции (интерфейсы C# могут определять операции!), и указания ограничения интерфейса для обобщенного класса. В любом случае первоначальный обзор построения специальных обобщенных типов завершен. Во время исследования типа делегата в главе 12 мы вновь обратимся к теме обобщений.

Резюме

Глава начиналась с рассмотрения необобщенных типов коллекций в пространствах имен `System.Collections` и `System.Collections.Specialized`, включая разнообразные проблемы, которые связаны со многими необобщенными контейнерами, в том числе отсутствие безопасности в отношении типов и накладные расходы времени выполнения в форме операций упаковки и распаковки. Как упоминалось, именно по этим причинам в современных приложениях .NET будут использоваться классы обобщенных коллекций из пространств имен `System.Collections.Generic` и `System.Collections.ObjectModel`.

Вы видели, что обобщенный элемент позволяет указывать заполнители (параметры типа), которые задаются во время создания объекта (или вызова в случае обобщенных методов). Хотя чаще всего вы будете просто применять обобщенные типы, предоставляемые библиотеками базовых классов .NET, также имеется возможность создавать собственные обобщенные типы (и обобщенные методы). При этом допускается указывать любое количество ограничений (с использованием ключевого слова `where`) для повышения уровня безопасности в отношении типов и гарантии того, что операции выполняются над типами *известного размера*, демонстрируя наличие определенных базовых возможностей.

В качестве финального замечания: не забывайте, что обобщения можно обнаружить во многих местах внутри библиотек базовых классов .NET. Здесь мы сосредоточились конкретно на обобщенных коллекциях. Тем не менее, по мере проработки материала оставшихся глав (и освоения платформы) вы наверняка найдете обобщенные классы, структуры и делегаты в том или ином пространстве имен. Кроме того, будьте готовы столкнуться с обобщенными членами в необобщенном классе!

ГЛАВА 11

Расширенные средства языка C#

В настоящей главе ваше понимание языка программирования C# будет углублено за счет исследования нескольких более сложных тем. Сначала вы узнаете, как реализовывать и применять *индексаторный метод*. Такой механизм C# позволяет строить специальные типы, которые предоставляют доступ к внутренним элементам с использованием синтаксиса, подобного синтаксису массивов. Вы научитесь перегружать разнообразные операции (+, –, <, > и т.д.) и создавать для своих типов специальные процедуры явного и неявного преобразования (а также ознакомитесь с причинами, по которым они могут понадобиться).

Затем будут обсуждаться темы, которые особенно полезны при работе с API-интерфейсами LINQ (хотя они применимы и за рамками контекста LINQ): расширяющие методы и анонимные типы.

В завершение главы вы узнаете, каким образом создавать контекст "небезопасного" кода, чтобы напрямую манипулировать неуправляемыми указателями. Хотя использование указателей в приложениях C# — довольно редкое явление, понимание того, как это делается, может пригодиться в определенных обстоятельствах, связанных со сложными сценариями взаимодействия.

Понятие индексаторных методов

Программистам хорошо знаком процесс доступа к индивидуальным элементам, содержащимся внутри простого массива, с применением операции индекса ([]). Вот пример:

```
// Организовать цикл по аргументам командной строки
// с использованием операции индекса.
for(int i = 0; i < args.Length; i++)
{
  Console.WriteLine("Args: {0}", args[i]);
}
// Объявить массив локальных целочисленных значений.
int[] myInts = { 10, 9, 100, 432, 9874};
// Применить операцию индекса для доступа к каждому элементу.
for(int j = 0; j < myInts.Length; j++)
{
  Console.WriteLine("Index {0} = {1} ", j, myInts[j]);
}
Console.ReadLine();
```

Приведенный код ни в коем случае не является чем-то совершенно новым. Но в языке C# предлагается возможность проектирования специальных классов и структур, которые могут индексироваться подобно стандартному массиву, за счет определения *индексаторного метода*. Такое языковое средство наиболее полезно при создании специальных классов коллекций (обобщенных или необобщенных).

Прежде чем выяснять, каким образом реализуется специальный индексатор, давайте начнем с того, что продемонстрируем его в действии. Пусть к специальному типу `PersonCollection`, разработанному в главе 10 (в проекте `IssuesWithNonGenericCollections`), добавлена поддержка индексаторного метода. Хотя сам индексатор пока не добавлен, давайте посмотрим, как он используется внутри нового проекта консольного приложения по имени `SimpleIndexer`:

```
using System;
using System.Collections.Generic;
using System.Data;
using SimpleIndexer;

// Индексаторы позволяют обращаться к элементам в стиле массива.
Console.WriteLine("***** Fun with Indexers *****\n");

PersonCollection myPeople = new PersonCollection();

// Добавить объекты с применением синтаксиса индексатора.
myPeople[0] = new Person("Homer", "Simpson", 40);
myPeople[1] = new Person("Marge", "Simpson", 38);
myPeople[2] = new Person("Lisa", "Simpson", 9);
myPeople[3] = new Person("Bart", "Simpson", 7);
myPeople[4] = new Person("Maggie", "Simpson", 2);

// Получить и отобразить элементы, используя индексатор.
for (int i = 0; i < myPeople.Count; i++)
{
  Console.WriteLine("Person number: {0}", i);          // номер лица
  Console.WriteLine("Name: {0} {1}",
    myPeople[i].FirstName, myPeople[i].LastName);      // имя и фамилия
  Console.WriteLine("Age: {0}", myPeople[i].Age);      // возраст
  Console.WriteLine();
}
```

Как видите, индексаторы позволяют манипулировать внутренней коллекцией подобъектов подобно стандартному массиву. Но тут возникает серьезный вопрос: каким образом сконфигурировать класс `PersonCollection` (или любой другой класс либо структуру) для поддержки такой функциональности? Индексатор представлен как слегка видоизмененное определение свойства C#. В своей простейшей форме индексатор создается с применением синтаксиса `this[]`. Ниже показано необходимое обновление класса `PersonCollection`:

```
using System.Collections;

namespace SimpleIndexer
{
  // Добавить индексатор к существующему определению класса.
  public class PersonCollection : IEnumerable
  {
    private ArrayList arPeople = new ArrayList();
    ...
```

```
    // Специальный индексатор для этого класса.
    public Person this[int index]
    {
      get => (Person)arPeople[index];
      set => arPeople.Insert(index, value);
    }
  }
}
```

Если не считать факт использования ключевого слова `this` с квадратными скобками, то индексатор похож на объявление любого другого свойства C#. Например, роль области `get` заключается в возвращении корректного объекта вызывающему коду. Здесь мы достигаем цели делегированием запроса к индексатору объекта `ArrayList`, т.к. данный класс также поддерживает индексатор. Область `set` контролирует добавление новых объектов `Person`, что достигается вызовом метода `Insert()` объекта `ArrayList`.

Индексаторы являются еще одной формой "синтаксического сахара", учитывая то, что такую же функциональность можно получить с применением "нормальных" открытых методов наподобие `AddPerson()` или `GetPerson()`. Тем не менее, поддержка индексаторных методов в специальных типах коллекций обеспечивает хорошую интеграцию с инфраструктурой библиотек базовых классов .NET Core.

Несмотря на то что создание индексаторных методов является вполне обычным явлением при построении специальных коллекций, не забывайте, что обобщенные типы предлагают такую функциональность в готовом виде. В следующем методе используется обобщенный список `List<T>` объектов `Person`. Обратите внимание, что индексатор `List<T>` можно просто применять непосредственно:

```
using System.Collections.Generic;
static void UseGenericListOfPeople()
{
  List<Person> myPeople = new List<Person>();
  myPeople.Add(new Person("Lisa", "Simpson", 9));
  myPeople.Add(new Person("Bart", "Simpson", 7));

  // Изменить первый объект лица с помощью индексатора.
  myPeople[0] = new Person("Maggie", "Simpson", 2);

  // Получить и отобразить каждый элемент, используя индексатор.
  for (int i = 0; i < myPeople.Count; i++)
  {
    Console.WriteLine("Person number: {0}", i);
    Console.WriteLine("Name: {0} {1}", myPeople[i].FirstName,
      myPeople[i].LastName);
    Console.WriteLine("Age: {0}", myPeople[i].Age);
    Console.WriteLine();
  }
}
```

Индексация данных с использованием строковых значений

В текущей версии класса `PersonCollection` определен индексатор, позволяющий вызывающему коду идентифицировать элементы с применением числовых значений. Однако вы должны понимать, что это не требование индексаторного метода. Предположим, что вы предпочитаете хранить объекты `Person`, используя тип

System.Collections.Generic.Dictionary<TKey, TValue>, а не ArrayList. Поскольку типы Dictionary разрешают доступ к содержащимся внутри них элементам с применением ключа (такого как фамилия лица), индексатор можно было бы определить следующим образом:

```
using System.Collections;
using System.Collections.Generic;

namespace SimpleIndexer
{
  public class PersonCollectionStringIndexer : IEnumerable
  {
    private Dictionary<string, Person> listPeople =
       new Dictionary<string, Person>();

    // Этот индексатор возвращает объект лица на основе строкового индекса.
    public Person this[string name]
    {
      get => (Person)listPeople[name];
      set => listPeople[name] = value;
    }

    public void ClearPeople()
    {
      listPeople.Clear();
    }

    public int Count => listPeople.Count;

    IEnumerator IEnumerable.GetEnumerator() => listPeople.GetEnumerator();
  }
}
```

Теперь вызывающий код способен взаимодействовать с содержащимися внутри объектами Person:

```
Console.WriteLine("***** Fun with Indexers *****\n");

PersonCollectionStringIndexer myPeopleStrings =
  new PersonCollectionStringIndexer();

myPeopleStrings["Homer"] =
  new Person("Homer", "Simpson", 40);
myPeopleStrings["Marge"] =
  new Person("Marge", "Simpson", 38);

// Получить объект лица Homer и вывести данные.
Person homer = myPeopleStrings["Homer"];
Console.ReadLine();
```

И снова, если бы обобщенный тип Dictionary<TKey, TValue> использовался напрямую, то функциональность индексаторного метода была бы получена в готовом виде без построения специального необобщенного класса, поддерживающего строковый индексатор. Тем не менее, имейте в виду, что тип данных любого индексатора будет основан на том, как поддерживающий тип коллекции позволяет вызывающему коду извлекать элементы.

Перегрузка индексаторных методов

Индексаторные методы могут быть перегружены в отдельном классе или структуре. Таким образом, если имеет смысл предоставить вызывающему коду возможность доступа к элементам с применением числового индекса *или* строкового значения, то в одном типе можно определить несколько индексаторов. Например, в ADO.NET (встроенный API-интерфейс .NET для доступа к базам данных) класс `DataSet` поддерживает свойство по имени `Tables`, которое возвращает строго типизированную коллекцию `DataTableCollection`. В свою очередь тип `DataTableCollection` определяет *три* индексатора для получения и установки объектов `DataTable` — по порядковой позиции, по дружественному строковому имени и по строковому имени с дополнительным пространством имен:

```
public sealed class DataTableCollection : InternalDataCollectionBase
{
  ...
  // Перегруженные индексаторы.
  public DataTable this[int index] { get; }
  public DataTable this[string name] { get; }
  public DataTable this[string name, string tableNamespace] { get; }
}
```

Поддержка индексаторных методов вполне обычна для типов в библиотеках базовых классов. Поэтому даже если текущий проект не требует построения специальных индексаторов для классов и структур, помните о том, что многие типы уже поддерживают такой синтаксис.

Многомерные индексаторы

Допускается также создавать индексаторный метод, который принимает несколько параметров. Предположим, что есть специальная коллекция, хранящая элементы в двумерном массиве. В таком случае индексаторный метод можно определить следующим образом:

```
public class SomeContainer
{
  private int[,] my2DintArray = new int[10, 10];

  public int this[int row, int column]
  { /* получить или установить значение в двумерном массиве */ }
}
```

Если только вы не строите высокоспециализированный класс коллекций, то вряд ли будете особо нуждаться в создании многомерного индексатора. Пример ADO.NET еще раз демонстрирует, насколько полезной может оказаться такая конструкция. Класс `DataTable` в ADO.NET по существу представляет собой коллекцию строк и столбцов, похожую на миллиметровку или на общую структуру электронной таблицы Microsoft Excel.

Хотя объекты `DataTable` обычно наполняются без вашего участия с использованием связанного "адаптера данных", в приведенном ниже коде показано, как вручную создать находящийся в памяти объект `DataTable`, который содержит три столбца (для имени, фамилии и возраста каждой записи). Обратите внимание на то, как после добавления одной строки в `DataTable` с помощью многомерного индексатора производится

обращение ко всем столбцам первой (и единственной) строки. (Если вы собираетесь следовать примеру, тогда импортируйте в файл кода пространство имен `System.Data`.)

```
static void MultiIndexerWithDataTable()
{
  // Создать простой объект DataTable с тремя столбцами.
  DataTable myTable = new DataTable();
  myTable.Columns.Add(new DataColumn("FirstName"));
  myTable.Columns.Add(new DataColumn("LastName"));
  myTable.Columns.Add(new DataColumn("Age"));

  // Добавить строку в таблицу.
  myTable.Rows.Add("Mel", "Appleby", 60);

  //Использовать многомерный индексатор для вывода деталей первой строки
  Console.WriteLine("First Name: {0}", myTable.Rows[0][0]);
  Console.WriteLine("Last Name: {0}", myTable.Rows[0][1]);
  Console.WriteLine("Age : {0}", myTable.Rows[0][2]);
}
```

Начиная с главы 21, мы продолжим рассмотрение ADO.NET, так что не переживайте, если что-то в приведенном выше коде выглядит незнакомым. Пример просто иллюстрирует, что индексаторные методы способны поддерживать множество измерений, а при правильном применении могут упростить взаимодействие с объектами, содержащимися в специальных коллекциях.

Определения индексаторов в интерфейсных типах

Индексаторы могут определяться в выбранном типе интерфейса .NET Core, чтобы позволить поддерживающим типам предоставлять специальные реализации. Ниже показан простой пример интерфейса, который задает протокол для получения строковых объектов с использованием числового индексатора:

```
public interface IStringContainer
{
  string this[int index] { get; set; }
}
```

При таком определении интерфейса любой класс или структура, которые его реализуют, должны поддерживать индексатор с чтением/записью, манипулирующий элементами с применением числового значения. Вот частичная реализация класса подобного вида:

```
class SomeClass : IStringContainer
{
  private List<string> myStrings = new List<string>();
  public string this[int index]
  {
    get => myStrings[index];
    set => myStrings.Insert(index, value);
  }
}
```

На этом первая крупная тема настоящей главы завершена. А теперь давайте перейдем к исследованиям языкового средства, которое позволяет строить специальные классы и структуры, уникальным образом реагирующие на внутренние операции C#. Итак, займемся концепцией *перегрузки операций*.

Понятие перегрузки операций

Как и любой язык программирования, C# имеет заготовленный набор лексем, используемых для выполнения базовых операций над встроенными типами. Например, вы знаете, что операция + может применяться к двум целым числам, чтобы получить большее целое число:

```
// Операция + с целыми числами.
int a = 100;
int b = 240;
int c = a + b;          // c теперь имеет значение 340
```

Опять-таки, здесь нет ничего нового, но задумывались ли вы когда-нибудь о том, что одну и ту же операцию + разрешено использовать с большинством встроенных типов данных C#? Скажем, взгляните на следующий код:

```
// Операция + со строками.
string s1 = "Hello";
string s2 = " World!";
string s3 = s1 + s2;    // s3 теперь имеет значение "Hello World!"
```

Операция + функционирует специфическим образом на основе типа предоставленных данных (в рассматриваемом случае строкового или целочисленного). Когда операция + применяется к числовым типам, в результате выполняется суммирование операндов, а когда к строковым типам — то конкатенация строк.

Язык C# дает возможность строить специальные классы и структуры, которые также уникально реагируют на один и тот же набор базовых лексем (вроде операции +). Хотя не каждая операция C# может быть перегружена, перегрузку допускают многие операции (табл. 11.1).

Таблица 11.1. Возможность перегрузки операций C#

Операция C#	Возможность перегрузки
+, –, !, ~, ++, --, true, false	Эти унарные операции могут быть перегружены. Язык C# требует совместной перегрузки true и false
+, –, *, /, %, &, |, ^, <<, >>	Эти бинарные операции могут быть перегружены
==, !=, <, >, <=, >=	Эти операции сравнения могут быть перегружены. Язык C# требует совместной перегрузки "похожих" операций (т.е. < и >, <= и >=, == и !=)
[]	Операция [] не может быть перегружена. Однако, как было показано ранее в главе, ту же самую функциональность обеспечивает индексатор
()	Операция () не может быть перегружена. Тем не менее, как вы увидите далее в главе, ту же самую функциональность предоставляют специальные методы преобразования
+=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Сокращенные операции присваивания не могут быть перегружены; однако вы получаете их автоматически, когда перегружаете соответствующие бинарные операции

Перегрузка бинарных операций

Чтобы проиллюстрировать процесс перегрузки бинарных операций, рассмотрим приведенный ниже простой класс `Point`, который определен в новом проекте консольного приложения по имени `OverloadedOps`:

```
using System;
namespace OverloadedOps
{
  // Простой будничный класс C#.
  public class Point
  {
    public int X {get; set;}
    public int Y {get; set;}

    public Point(int xPos, int yPos)
    {
      X = xPos;
      Y = yPos;
    }

    public override string ToString()
      => $"[{this.X}, {this.Y}]";
  }
}
```

Рассуждая логически, суммирование объектов `Point` имеет смысл. Например, сложение двух переменных `Point` должно давать новый объект `Point` с просуммированными значениями свойств `X` и `Y`. Конечно, полезно также и вычитать один объект `Point` из другого. В идеале желательно иметь возможность записи примерно такого кода:

```
using System;
using OverloadedOps;

// Сложение и вычитание двух точек?
Console.WriteLine("***** Fun with Overloaded Operators *****\n");

// Создать две точки.
Point ptOne = new Point(100, 100);
Point ptTwo = new Point(40, 40);
Console.WriteLine("ptOne = {0}", ptOne);
Console.WriteLine("ptTwo = {0}", ptTwo);

// Сложить две точки, чтобы получить большую точку?
Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

// Вычесть одну точку из другой, чтобы получить меньшую?
Console.WriteLine("ptOne - ptTwo: {0} ", ptOne - ptTwo);
Console.ReadLine();
```

Тем не менее, с существующим видом класса `Point` вы получите ошибки на этапе компиляции, потому что типу `Point` не известно, как реагировать на операцию `+` или `-`. Для оснащения специального типа способностью уникально реагировать на встроенные операции язык C# предлагает ключевое слово `operator`, которое может использоваться только в сочетании с ключевым словом `static`. При перегрузке бинарной операции (такой как `+` и `-`) вы чаще всего будете передавать два аргумента того же типа, что и класс, определяющий операцию (`Point` в этом примере):

```
// Более интеллектуальный тип Point.
public class Point
{
  ...
  // Перегруженная операция +.
  public static Point operator + (Point p1, Point p2)
    => new Point(p1.X + p2.X, p1.Y + p2.Y);

  // Перегруженная операция -.
  public static Point operator - (Point p1, Point p2)
    => new Point(p1.X - p2.X, p1.Y - p2.Y);
}
```

Логика, положенная в основу операции +, предусматривает просто возвращение нового объекта `Point`, основанного на сложении соответствующих полей входных параметров `Point`. Таким образом, когда вы пишете `p1 + p2`, "за кулисами" происходит следующий скрытый вызов статического метода `operator +`:

```
// Псевдокод: Point p3 = Point.operator+ (p1, p2)
Point p3 = p1 + p2;
```

Аналогично выражение `p1 - p2` отображается так:

```
// Псевдокод: Point p4 = Point.operator- (p1, p2)
Point p4 = p1 - p2;
```

После произведенной модификации типа `Point` программа скомпилируется и появится возможность сложения и вычитания объектов `Point`, что подтверждает представленный далее вывод:

```
***** Fun with Overloaded Operators *****
ptOne = [100, 100]
ptTwo = [40, 40]
ptOne + ptTwo: [140, 140]
ptOne - ptTwo: [60, 60]
```

При перегрузке бинарной операции передавать ей два параметра того же самого типа не обязательно. Если это имеет смысл, тогда один из аргументов может относиться к другому типу. Например, ниже показана перегруженная операция +, которая позволяет вызывающему коду получить новый объект `Point` на основе числовой коррекции:

```
public class Point
{
  ...
  public static Point operator + (Point p1, int change)
    => new Point(p1.X + change, p1.Y + change);

  public static Point operator + (int change, Point p1)
    => new Point(p1.X + change, p1.Y + change);
}
```

Обратите внимание, что если вы хотите передавать аргументы в любом порядке, то потребуется реализовать *обе* версии метода (т.е. нельзя просто определить один из методов и рассчитывать, что компилятор автоматически будет поддерживать другой). Теперь новые версии операции + можно применять следующим образом:

```
// Выводит [110, 110].
Point biggerPoint = ptOne + 10;
Console.WriteLine("ptOne + 10 = {0}", biggerPoint);
// Выводит [120, 120].
Console.WriteLine("10 + biggerPoint = {0}", 10 + biggerPoint);
Console.WriteLine();
```

А как насчет операций += и -=?

Если до перехода на C# вы имели дело с языком C++, тогда вас может удивить отсутствие возможности перегрузки операций сокращенного присваивания (+=, -+ и т.д.). Не беспокойтесь. В C# операции сокращенного присваивания автоматически эмулируются в случае перегрузки связанных бинарных операций. Таким образом, если в классе `Point` уже перегружены операции + и -, то можно написать приведенный далее код:

```
// Перегрузка бинарных операций автоматически обеспечивает
// перегрузку сокращенных операций.
...
// Автоматически перегруженная операция +=.
Point ptThree = new Point(90, 5);
Console.WriteLine("ptThree = {0}", ptThree);
Console.WriteLine("ptThree += ptTwo: {0}", ptThree += ptTwo);
// Автоматически перегруженная операция -=.
Point ptFour = new Point(0, 500);
Console.WriteLine("ptFour = {0}", ptFour);
Console.WriteLine("ptFour -= ptThree: {0}", ptFour -= ptThree);
Console.ReadLine();
```

Перегрузка унарных операций

В языке C# также разрешено перегружать унарные операции, такие как ++ и --. При перегрузке унарной операции также должно использоваться ключевое слово `static` с ключевым словом `operator`, но в этом случае просто передается единственный параметр того же типа, что и класс или структура, где операция определена. Например, дополните реализацию `Point` следующими перегруженными операциями:

```
public class Point
{
  ...
  // Добавить 1 к значениям X/Y входного объекта Point.
  public static Point operator ++(Point p1)
    => new Point(p1.X+1, p1.Y+1);

  // Вычесть 1 из значений X/Y входного объекта Point.
  public static Point operator --(Point p1)
    => new Point(p1.X-1, p1.Y-1);
}
```

В результате появляется возможность инкрементировать и декрементировать значения `X` и `Y` класса `Point`:

```
...
// Применение унарных операций ++ и -- к объекту Point.
Point ptFive = new Point(1, 1);
Console.WriteLine("++ptFive = {0}", ++ptFive);  // [2, 2]
Console.WriteLine("--ptFive = {0}", --ptFive);  // [1, 1]
```

```
// Применение тех же операций в виде постфиксного инкремента/декремента.
Point ptSix = new Point(20, 20);
Console.WriteLine("ptSix++ = {0}", ptSix++);    // [20, 20]
Console.WriteLine("ptSix-- = {0}", ptSix--);    // [21, 21]
Console.ReadLine();
```

В предыдущем примере кода специальные операции ++ и -- применяются двумя разными способами. В языке C++ допускается перегружать операции префиксного и постфиксного инкремента/декремента по отдельности. В C# это невозможно. Однако возвращаемое значение инкремента/декремента автоматически обрабатывается корректно (т.е. для перегруженной операции ++ выражение pt++ дает значение неизменного объекта, в то время как результатом ++pt будет новое значение, устанавливаемое перед использованием в выражении).

Перегрузка операций эквивалентности

Как упоминалось в главе 6, метод System.Object.Equals() может быть перегружен для выполнения сравнений на основе значений (а не ссылок) между ссылочными типами. Если вы решили переопределить Equals() (часто вместе со связанным методом System.Object.GetHashCode()), то легко переопределите и операции проверки эквивалентности (== и !=). Взгляните на обновленный тип Point:

```
// В данной версии типа Point также перегружены операции == и !=.
public class Point
{
  ...
  public override bool Equals(object o)
    => o.ToString() == this.ToString();

  public override int GetHashCode()
    => this.ToString().GetHashCode();

  // Теперь перегрузить операции == и !=.
  public static bool operator ==(Point p1, Point p2)
    => p1.Equals(p2);

  public static bool operator !=(Point p1, Point p2)
    => !p1.Equals(p2);
}
```

Обратите внимание, что для выполнения всей работы в реализациях операций == и != просто вызывается перегруженный метод Equals(). Вот как теперь можно применять класс Point:

```
// Использование перегруженных операций эквивалентности.
...
Console.WriteLine("ptOne == ptTwo : {0}", ptOne == ptTwo);
Console.WriteLine("ptOne != ptTwo : {0}", ptOne != ptTwo);
Console.ReadLine();
```

Как видите, сравнение двух объектов с использованием хорошо знакомых операций == и != выглядит намного интуитивно понятнее, чем вызов метода Object.Equals(). При перегрузке операций эквивалентности для определенного класса имейте в виду, что C# требует, чтобы в случае перегрузки операции == *обязательно* перегружалась также и операция != (компилятор напомнит, если вы забудете это сделать).

Перегрузка операций сравнения

В главе 8 было показано, каким образом реализовывать интерфейс `IComparable` для сравнения двух похожих объектов. В действительности для того же самого класса можно также перегрузить операции сравнения (<, >, <= и >=). Как и в случае операций эквивалентности, язык C# требует, чтобы при перегрузке операции < обязательно перегружалась также операция >. Если класс `Point` перегружает указанные операции сравнения, тогда пользователь объекта может сравнивать объекты `Point`:

```
// Использование перегруженных операций < и >.
...
Console.WriteLine("ptOne < ptTwo : {0}", ptOne < ptTwo);
Console.WriteLine("ptOne > ptTwo : {0}", ptOne > ptTwo);
Console.ReadLine();
```

Когда интерфейс `IComparable` (или, что еще лучше, его обобщенный эквивалент) реализован, перегрузка операций сравнения становится тривиальной. Вот модифицированное определение класса:

```
// Объекты Point также можно сравнивать посредством операций сравнения.
public class Point : IComparable<Point>
{
  ...
  public int CompareTo(Point other)
  {
    if (this.X > other.X && this.Y > other.Y)
    {
      return 1;
    }
    if (this.X < other.X && this.Y < other.Y)
    {
      return -1;
    }
    return 0;
  }
  public static bool operator <(Point p1, Point p2)
    => p1.CompareTo(p2) < 0;
  public static bool operator >(Point p1, Point p2)
    => p1.CompareTo(p2) > 0;
  public static bool operator <=(Point p1, Point p2)
    => p1.CompareTo(p2) <= 0;
  public static bool operator >=(Point p1, Point p2)
    => p1.CompareTo(p2) >= 0;
}
```

Финальные соображения относительно перегрузки операций

Как уже упоминалось, язык C# предоставляет возможность построения типов, которые могут уникальным образом реагировать на разнообразные встроенные хорошо известные операции. Перед добавлением поддержки такого поведения в классы вы должны удостовериться в том, что операции, которые планируется перегружать, имеют какой-нибудь смысл в реальности.

Например, пусть перегружена операция умножения для класса MiniVan, представляющего минивэн. Что по своей сути будет означать перемножение двух объектов MiniVan? В нем нет особого смысла. На самом деле коллеги по команде даже могут быть озадачены, когда увидят следующее применение класса MiniVan:

```
// Что?! Понять это непросто...
MiniVan newVan = myVan * yourVan;
```

Перегрузка операций обычно полезна только при построении атомарных типов данных. Векторы, матрицы, текст, точки, фигуры, множества и т.п. будут подходящими кандидатами на перегрузку операций, но люди, менеджеры, автомобили, подключения к базе данных и веб-страницы — нет. В качестве эмпирического правила запомните, что если перегруженная операция *затрудняет* понимание пользователем функциональности типа, то не перегружайте ее. Используйте такую возможность с умом.

Понятие специальных преобразований типов

Давайте теперь обратимся к теме, тесно связанной с перегрузкой операций, а именно — к специальным преобразованиям типов. Чтобы заложить фундамент для последующего обсуждения, кратко вспомним понятие явных и неявных преобразований между числовыми данными и связанными типами классов.

Повторение: числовые преобразования

В терминах встроенных числовых типов (sbyte, int, float и т.д.) *явное преобразование* требуется, когда вы пытаетесь сохранить большее значение в контейнере меньшего размера, т.к. подобное действие может привести к утере данных. По существу тем самым вы сообщаете компилятору, что отдаете себе отчет в том, что делаете. И наоборот — *неявное преобразование* происходит автоматически, когда вы пытаетесь поместить меньший тип в больший целевой тип, что не должно вызвать потерю данных:

```
int a = 123;
long b = a;         // Неявное преобразование из int в long.
int c = (int) b;    // Явное преобразование из long в int.
```

Повторение: преобразования между связанными типами классов

В главе 6 было показано, что типы классов могут быть связаны классическим наследованием (отношение "является"). В таком случае процесс преобразования C# позволяет осуществлять приведение вверх и вниз по иерархии классов. Например, производный класс всегда может быть неявно приведен к базовому классу. Тем не менее, если вы хотите сохранить объект базового класса в переменной производного класса, то должны выполнить явное приведение:

```
// Два связанных типа классов.
class Base{}
class Derived : Base{}

// Неявное приведение производного класса к базовому.
Base myBaseType;
myBaseType = new Derived();

// Для сохранения ссылки на базовый класс в переменной
// производного класса требуется явное преобразование.
Derived myDerivedType = (Derived)myBaseType;
```

Продемонстрированное явное приведение работает из-за того, что классы Base и Derived связаны классическим наследованием, а объект myBaseType создан как экземпляр Derived. Однако если myBaseType является экземпляром Base, тогда приведение вызывает генерацию исключения InvalidCastException. При наличии сомнений по поводу успешности приведения вы должны использовать ключевое слово as, как обсуждалось в главе 6. Ниже показан переделанный пример:

```
// Неявное приведение производного класса к базовому.
Base myBaseType2 = new();
// Сгенерируется исключение InvalidCastException:
// Derived myDerivedType2 = (Derived)myBaseType2 as Derived;
// Исключения нет, myDerivedType2 равен null:
Derived myDerivedType2 = myBaseType2 as Derived;
```

Но что если есть два типа классов в *разных иерархиях* без общего предка (кроме System.Object), которые требуют преобразований? Учитывая, что они не связаны классическим наследованием, типичные операции приведения здесь не помогут (и вдобавок компилятор сообщит об ошибке).

В качестве связанного замечания обратимся к типам значений (структурам). Предположим, что имеются две структуры с именами Square и Rectangle. Поскольку они не могут задействовать классическое наследование (т.к. запечатаны), не существует естественного способа выполнить приведение между этими по внешнему виду связанными типами.

Несмотря на то что в структурах можно было бы создать вспомогательные методы (наподобие Rectangle.ToSquare()), язык C# позволяет строить специальные процедуры преобразования, которые дают типам возможность реагировать на операцию приведения (). Следовательно, если корректно сконфигурировать структуры, тогда для явного преобразования между ними можно будет применять такой синтаксис:

```
// Преобразовать Rectangle в Square!
Rectangle rect = new Rectangle
{
  Width = 3;
  Height = 10;
}
Square sq = (Square)rect;
```

Создание специальных процедур преобразования

Начните с создания нового проекта консольного приложения по имени CustomConversions. В языке C# предусмотрены два ключевых слова, explicit и implicit, которые можно использовать для управления тем, как типы должны реагировать на попытку преобразования. Предположим, что есть следующие определения структур:

```
using System;

namespace CustomConversions
{
  public struct Rectangle
  {
    public int Width {get; set;}
    public int Height {get; set;}
```

```
    public Rectangle(int w, int h)
    {
      Width = w;
      Height = h;
    }
    public void Draw()
    {
      for (int i = 0; i < Height; i++)
      {
        for (int j = 0; j < Width; j++)
        {
          Console.Write("*");
        }
        Console.WriteLine();
      }
    }
    public override string ToString()
      => $"[Width = {Width}; Height = {Height}]";
  }
}
using System;
namespace CustomConversions
{
  public struct Square
  {
    public int Length {get; set;}
    public Square(int l) : this()
    {
      Length = l;
    }
    public void Draw()
    {
      for (int i = 0; i < Length; i++)
      {
        for (int j = 0; j < Length; j++)
        {
          Console.Write("*");
        }
        Console.WriteLine();
      }
    }
    public override string ToString() => $"[Length = {Length}]";
    // Rectangle можно явно преобразовывать в Square.
    public static explicit operator Square(Rectangle r)
    {
      Square s = new Square {Length = r.Height};
      return s;
    }
  }
}
```

Обратите внимание, что в текущей версии типа `Square` определена явная операция преобразования. Подобно перегрузке операций процедуры преобразования используют ключевое слово `operator` в сочетании с ключевым словом `explicit` или `implicit` и должны быть определены как `static`. Входным параметром является сущность, *из* которой выполняется преобразование, а типом операции — сущность, *в* которую производится преобразование.

В данном случае предположение заключается в том, что квадрат (будучи геометрической фигурой с четырьмя сторонами равной длины) может быть получен из высоты прямоугольника. Таким образом, вот как преобразовать `Rectangle` в `Square`:

```
using System;
using CustomConversions;

Console.WriteLine("***** Fun with Conversions *****\n");

// Создать экземпляр Rectangle.
Rectangle r = new Rectangle(15, 4);
Console.WriteLine(r.ToString());
r.Draw();

Console.WriteLine();

// Преобразовать r в Square на основе высоты Rectangle.
Square s = (Square)r;
Console.WriteLine(s.ToString());
s.Draw();
Console.ReadLine();
```

Ниже показан вывод:

```
***** Fun with Conversions *****

[Width = 15; Height = 4]
***************
***************
***************
***************

[Length = 4]
****
****
****
****
```

Хотя преобразование `Rectangle` в `Square` в пределах той же самой области действия может быть не особенно полезным, взгляните на следующий метод, который спроектирован для приема параметров типа `Square`:

```
// Этот метод требует параметр типа Square.
static void DrawSquare(Square sq)
{
  Console.WriteLine(sq.ToString());
  sq.Draw();
}
```

Благодаря наличию операции явного преобразования в типе `Square` методу `DrawSquare()` на обработку можно передавать типы `Rectangle`, применяя явное приведение:

```
...
// Преобразовать Rectangle в Square для вызова метода.
Rectangle rect = new Rectangle(10, 5);
DrawSquare((Square)rect);
Console.ReadLine();
```

Дополнительные явные преобразования для типа Square

Теперь, когда экземпляры Rectangle можно явно преобразовывать в экземпляры Square, давайте рассмотрим несколько дополнительных явных преобразований. Учитывая, что квадрат симметричен по всем сторонам, полезно предусмотреть процедуру преобразования, которая позволит вызывающему коду привести целочисленный тип к типу Square (который, естественно, будет иметь длину стороны, равную переданному целочисленному значению). А что если вы захотите модифицировать еще и Square так, чтобы вызывающий код мог выполнять приведение *из* Square в int? Вот как выглядит логика вызова:

```
...
// Преобразование int в Square.
Square sq2 = (Square)90;
Console.WriteLine("sq2 = {0}", sq2);

// Преобразование Square в int.
int side = (int)sq2;
Console.WriteLine("Side length of sq2 = {0}", side);
Console.ReadLine();
```

Ниже показаны изменения, внесенные в структуру Square:

```
public struct Square
{
  ...
  public static explicit operator Square(int sideLength)
  {
    Square newSq = new Square {Length = sideLength};
    return newSq;
  }
  public static explicit operator int (Square s) => s.Length;
}
```

По правде говоря, преобразование Square в int может показаться не слишком интуитивно понятной (или полезной) операцией (скорее всего, вы просто будете передавать нужные значения конструктору). Тем не менее, оно указывает на важный факт, касающийся процедур специальных преобразований: компилятор не беспокоится о том, из чего и во что происходит преобразование, до тех пор, пока вы пишете синтаксически корректный код.

Таким образом, как и с перегрузкой операций, *возможность* создания операции явного приведения для заданного типа вовсе не означает *необходимость* ее создания. Обычно этот прием будет наиболее полезным при создании типов структур, учитывая, что они не могут принимать участие в классическом наследовании (где приведение обеспечивается автоматически).

Определение процедур неявного преобразования

До сих пор мы создавали различные специальные операции *явного* преобразования. Но что насчет следующего *неявного* преобразования?

```
...
Square s3 = new Square {Length = 83};

// Попытка сделать неявное приведение?
Rectangle rect2 = s3;

Console.ReadLine();
```

Данный код не скомпилируется, т.к. вы не предоставили процедуру неявного преобразования для типа `Rectangle`. Ловушка здесь вот в чем: определять одновременно функции явного и неявного преобразования не разрешено, если они не различаются по типу возвращаемого значения или по списку параметров. Это может показаться ограничением; однако вторая ловушка связана с тем, что когда тип определяет процедуру *неявного* преобразования, то вызывающий код вполне законно может использовать синтаксис *явного* приведения!

Запутались? Чтобы прояснить ситуацию, давайте добавим к структуре `Rectangle` процедуру неявного преобразования с применением ключевого слова `implicit` (обратите внимание, что в показанном ниже коде предполагается, что ширина результирующего прямоугольника вычисляется умножением стороны квадрата на 2):

```
public struct Rectangle
{
  ...
  public static implicit operator Rectangle(Square s)
  {
    Rectangle r = new Rectangle
    {
      Height = s.Length,
      Width = s.Length * 2   // Предположим, что ширина нового
                             // квадрата будет равна (Length x 2).
    };
    return r;
  }
}
```

После такой модификации можно выполнять преобразование между типами:

```
...
// Неявное преобразование работает!
Square s3 = new Square { Length= 7};

Rectangle rect2 = s3;
Console.WriteLine("rect2 = {0}", rect2);

// Синтаксис явного приведения также работает!
Square s4 = new Square {Length = 3};
Rectangle rect3 = (Rectangle)s4;

Console.WriteLine("rect3 = {0}", rect3);
Console.ReadLine();
```

На этом обзор определения операций специального преобразования завершен. Как и с перегруженными операциями, помните о том, что данный фрагмент синтаксиса

представляет собой просто сокращенное обозначение для "нормальных" функций-членов и потому всегда необязателен. Тем не менее, в случае правильного использования специальные структуры могут применяться более естественным образом, поскольку будут трактоваться как настоящие типы классов, связанные наследованием.

Понятие расширяющих методов

В версии .NET 3.5 появилась концепция *расширяющих методов*, которая позволила добавлять новые методы или свойства к классу либо структуре, не модифицируя исходный тип непосредственно. Когда такой прием может оказаться полезным? Рассмотрим следующие ситуации.

Пусть есть класс, находящийся в производстве. Со временем выясняется, что имеющийся класс должен поддерживать несколько новых членов. Изменение текущего определения класса напрямую сопряжено с риском нарушения обратной совместимости со старыми кодовыми базами, использующими его, т.к. они могут не скомпилироваться с последним улучшенным определением класса. Один из способов обеспечения обратной совместимости предусматривает создание нового класса, производного от существующего, но тогда придется сопровождать два класса. Как все мы знаем, сопровождение кода является самой скучной частью деятельности разработчика программного обеспечения.

Представим другую ситуацию. Предположим, что имеется структура (или, может быть, запечатанный класс), и необходимо добавить новые члены, чтобы получить полиморфное поведение в рамках системы. Поскольку структуры и запечатанные классы не могут быть расширены, единственный выбор заключается в том, чтобы добавить желаемые члены к типу, снова рискуя нарушить обратную совместимость!

За счет применения расширяющих методов появляется возможность модифицировать типы, не создавая подклассов и не изменяя код типа напрямую. Загвоздка в том, что новая функциональность предлагается типом, только если в текущем проекте будут присутствовать ссылки на расширяющие методы.

Определение расширяющих методов

Первое ограничение, связанное с расширяющими методами, состоит в том, что они должны быть определены внутри статического класса (см. главу 5), а потому каждый расширяющий метод должен объявляться с ключевым словом `static`. Вторая проблема в том, что все расширяющие методы помечаются как таковые посредством ключевого слова `this` в качестве модификатора первого (и только первого) параметра заданного метода. Параметр, помеченный с помощью `this`, представляет расширяемый элемент.

В целях иллюстрации создайте новый проект консольного приложения под названием `ExtensionMethods`. Предположим, что создается класс по имени `MyExtensions`, в котором определены два расширяющих метода. Первый расширяющий метод позволяет объекту любого типа взаимодействовать с новым методом `DisplayDefiningAssembly()`, который использует типы из пространства имен `System.Reflection` для отображения имени сборки, содержащей данный тип.

На заметку! API-интерфейс рефлексии формально рассматривается в главе 17. Если эта тема для вас нова, тогда просто запомните, что рефлексия позволяет исследовать структуру сборок, типов и членов типов во время выполнения.

Второй расширяющий метод по имени `ReverseDigits()` позволяет любому значению типа `int` получить новую версию самого себя с обратным порядком следования цифр. Например, если целочисленное значение `1234` вызывает `ReverseDigits()`, то в результате возвратится `4321`. Взгляните на следующую реализацию класса (не забудьте импортировать пространство имен `System.Reflection`):

```
using System;
using System.Reflection;

namespace MyExtensionMethods
{
  static class MyExtensions
  {
    // Этот метод позволяет объекту любого типа
    // отобразить сборку, в которой он определен.
    public static void DisplayDefiningAssembly(this object obj)
    {
      Console.WriteLine("{0} lives here: => {1}\n", obj.GetType().Name,
        Assembly.GetAssembly(obj.GetType()).GetName().Name);
    }

    // Этот метод позволяет любому целочисленному значению изменить
    // порядок следования десятичных цифр на обратный.
    // Например, для 56 возвратится 65.
    public static int ReverseDigits(this int i)
    {
      // Транслировать int в string и затем получить все его символы.
      char[] digits = i.ToString().ToCharArray();

      // Изменить порядок следования элементов массива.
      Array.Reverse(digits);

      // Поместить обратно в строку.
      string newDigits = new string(digits);

      // Возвратить модифицированную строку как int.
      return int.Parse(newDigits);
    }
  }
}
```

Снова обратите внимание на то, что первый параметр каждого расширяющего метода снабжен ключевым словом `this`, находящимся перед определением типа параметра. Первый параметр расширяющего метода всегда представляет расширяемый тип. Учитывая, что метод `DisplayDefiningAssembly()` был прототипирован для расширения `System.Object`, этот новый член теперь присутствует в каждом типе, поскольку `Object` является родительским для всех типов платформы .NET Core. Однако метод `ReverseDigits()` прототипирован для расширения только целочисленных типов, и потому если к нему обращается какое-то другое значение, то возникнет ошибка на этапе компиляции.

На заметку! Запомните, что каждый расширяющий метод может иметь множество параметров, но только первый параметр разрешено помечать посредством `this`. Дополнительные параметры будут трактоваться как нормальные входные параметры, применяемые методом.

Вызов расширяющих методов

Располагая созданными расширяющими методами, рассмотрим следующий код, в котором они используются с разнообразными типами из библиотек базовых классов:

```
using System;
using MyExtensionMethods;

Console.WriteLine("***** Fun with Extension Methods *****\n");

// В int появилась новая отличительная черта!
int myInt = 12345678;
myInt.DisplayDefiningAssembly();

// То же и в DataSet!
System.Data.DataSet d = new System.Data.DataSet();
d.DisplayDefiningAssembly();

// И в SoundPlayer!
System.Media.SoundPlayer sp = new System.Media.SoundPlayer();
sp.DisplayDefiningAssembly();

// Использовать новую функциональность int.
Console.WriteLine("Value of myInt: {0}", myInt);
Console.WriteLine("Reversed digits of myInt: {0}", myInt.ReverseDigits());

Console.ReadLine();
```

Ниже показан вывод:

```
***** Fun with Extension Methods *****

Int32 lives here: => System.Private.CoreLib

DataSet lives here: => System.Data.Common

Value of myInt: 12345678
Reversed digits of myInt: 87654321
```

Импортирование расширяющих методов

Когда определяется класс, содержащий расширяющие методы, он вне всяких сомнений будет принадлежать какому-то пространству имен. Если это пространство имен отличается от пространства имен, где расширяющие методы применяются, тогда придется использовать ключевое слово `using` языка C#, которое позволит файлу кода иметь доступ ко всем расширяющим методам интересующего типа. Об этом важно помнить, потому что если явно не импортировать корректное пространство имен, то в таком файле кода C# расширяющие методы будут недоступными.

Хотя на первый взгляд может показаться, что расширяющие методы глобальны по своей природе, на самом деле они ограничены пространствами имен, где определены, или пространствами имен, которые их импортируют. Вспомните, что вы поместили класс `MyExtensions` в пространство имен `MyExtensionMethods`, как показано ниже:

```
namespace MyExtensionMethods
{
  static class MyExtensions
  {
    ...
  }
}
```

Для использования расширяющих методов класса `MyExtensions` необходимо явно импортировать пространство имен `MyExtensionMethods`, как делалось в рассмотренных ранее примерах операторов верхнего уровня.

Расширение типов, реализующих специфичные интерфейсы

К настоящему моменту вы видели, как расширять классы (и косвенно структуры, которые следуют тому же синтаксису) новой функциональностью через расширяющие методы. Также есть возможность определить расширяющий метод, который способен расширять только класс или структуру, реализующую корректный интерфейс. Например, можно было бы заявить следующее: если класс или структура реализует интерфейс `IEnumerable<T>`, тогда этот тип получит новые члены. Разумеется, вполне допустимо требовать, чтобы тип поддерживал вообще любой интерфейс, включая ваши специальные интерфейсы.

В качестве примера создайте новый проект консольного приложения по имени `InterfaceExtensions`. Цель здесь заключается в том, чтобы добавить новый метод к любому типу, который реализует интерфейс `IEnumerable`, что охватывает все массивы и многие классы необобщенных коллекций (вспомните из главы 10, что обобщенный интерфейс `IEnumerable<T>` расширяет необобщенный интерфейс `IEnumerable`). Добавьте к проекту следующий расширяющий класс:

```
using System;

namespace InterfaceExtensions
{
  static class AnnoyingExtensions
  {
    public static void PrintDataAndBeep(
        this System.Collections.IEnumerable iterator)
    {
      foreach (var item in iterator)
      {
        Console.WriteLine(item);
        Console.Beep();
      }
    }
  }
}
```

Поскольку метод `PrintDataAndBeep()` может использоваться любым классом или структурой, реализующей интерфейс `IEnumerable`, мы можем протестировать его с помощью такого кода:

```
using System;
using System.Collections.Generic;
using InterfaceExtensions;

Console.WriteLine("***** Extending Interface Compatible Types *****\n");

// System.Array реализует IEnumerable!
string[] data = { "Wow", "this", "is", "sort", "of", "annoying",
                  "but", "in", "a", "weird", "way", "fun!"};
data.PrintDataAndBeep();

Console.WriteLine();
```

```
// List<T> реализует IEnumerable!
List<int> myInts = new List<int>() {10, 15, 20};
myInts.PrintDataAndBeep();

Console.ReadLine();
```

На этом исследование расширяющих методов C# завершено. Помните, что данное языковое средство полезно, когда необходимо расширить функциональность типа, но вы не хотите создавать подклассы (или не можете, если тип запечатан) в целях обеспечения полиморфизма. Как вы увидите позже, расширяющие методы играют ключевую роль в API-интерфейсах LINQ. На самом деле вы узнаете, что в API-интерфейсах LINQ одним из самых часто расширяемых элементов является класс или структура, реализующая обобщенную версию интерфейса `IEnumerable`.

Поддержка расширяющего метода `GetEnumerator()` (нововведение в версии 9.0)

До выхода версии C# 9.0 для применения оператора `foreach` с экземплярами класса в этом классе нужно было напрямую определять метод `GetEnumerator()`. Начиная с версии C# 9.0, оператор `foreach` исследует расширяющие методы класса и в случае, если обнаруживает метод `GetEnumerator()`, то использует его для получения реализации `IEnumerator`, относящейся к данному классу. Чтобы удостовериться в сказанном, добавьте новый проект консольного приложения по имени `ForEachWithExtensionMethods` и поместите в него упрощенные версии классов `Car` и `Garage` из главы 8:

```
// Car.cs
using System;

namespace ForEachWithExtensionMethods
{
  class Car
  {
    // Свойства класса Car.
    public int CurrentSpeed {get; set;} = 0;
    public string PetName {get; set;} = "";

    // Конструкторы.
    public Car() {}
    public Car(string name, int speed)
    {
      CurrentSpeed = speed;
      PetName = name;
    }

    // Выяснить, не перегрелся ли двигатель Car.
  }
}

// Garage.cs
namespace ForEachWithExtensionMethods
{
  class Garage
  {
    public Car[] CarsInGarage { get; set; }

    // При запуске заполнить несколькими объектами Car.
```

```
    public Garage()
    {
      CarsInGarage = new Car[4];
      CarsInGarage[0] = new Car("Rusty", 30);
      CarsInGarage[1] = new Car("Clunker", 55);
      CarsInGarage[2] = new Car("Zippy", 30);
      CarsInGarage[3] = new Car("Fred", 30);
    }
  }
}
```

Обратите внимание, что класс `Garage` не реализует интерфейс `IEnumerable` и не имеет метода `GetEnumerator()`. Метод `GetEnumerator()` добавляется через показанный ниже класс `GarageExtensions`:

```
using System.Collections;

namespace ForEachWithExtensionMethods
{
  static class GarageExtensions
  {
    public static IEnumerator GetEnumerator(this Garage g)
      => g.CarsInGarage.GetEnumerator();
  }
}
```

Код для тестирования этого нового средства будет таким же, как код, который применялся для тестирования метода `GetEnumerator()` в главе 8. Модифицируйте файл `Program.cs` следующим образом:

```
using System;
using ForEachWithExtensionMethods;

Console.WriteLine("***** Support for Extension Method GetEnumerator
*****\n");
Garage carLot = new Garage();

// Проход по всем объектам Car в коллекции?
foreach (Car c in carLot)
{
  Console.WriteLine("{0} is going {1} MPH",
    c.PetName, c.CurrentSpeed);
}
```

Вы увидите, что код работает, успешно выводя на консоль список объектов автомобилей и скоростей их движения:

```
***** Support for Extension Method GetEnumerator *****

Rusty is going 30 MPH
Clunker is going 55 MPH
Zippy is going 30 MPH
Fred is going 30 MPH
```

На заметку! Потенциальный недостаток нового средства заключается в том, что теперь с оператором `foreach` могут использоваться даже те классы, которые для этого не предназначались.

Понятие анонимных типов

Программистам на объектно-ориентированных языках хорошо известны преимущества определения классов для представления состояния и функциональности заданного элемента, который требуется моделировать. Всякий раз, когда необходимо определить класс, предназначенный для многократного применения и предоставляющий обширную функциональность через набор методов, событий, свойств и специальных конструкторов, устоявшаяся практика предусматривает создание нового класса C#.

Тем не менее, возникают и другие ситуации, когда желательно определять класс просто в целях моделирования набора инкапсулированных (и каким-то образом связанных) элементов данных безо всяких ассоциированных методов, событий или другой специализированной функциональности. Кроме того, что если такой тип должен использоваться только небольшим набором методов внутри программы? Было бы довольно утомительно строить полное определение класса вроде показанного ниже, если хорошо известно, что класс будет применяться только в нескольких местах. Чтобы подчеркнуть данный момент, вот примерный план того, что может понадобиться делать, когда нужно создать "простой" тип данных, который следует обычной семантике на основе значений:

```
class SomeClass
{
  // Определить набор закрытых переменных-членов...

  // Создать свойство для каждой закрытой переменной-члена...

  // Переопределить метод ToString() для учета основных
  // переменных-членов...

  // Переопределить методы GetHashCode() и Equals() для работы
  // с эквивалентностью на основе значений...
}
```

Как видите, задача не обязательно оказывается настолько простой. Вам потребуется не только написать большой объем кода, но еще и сопровождать дополнительный класс в системе. Для временных данных подобного рода было бы удобно формировать специальный тип на лету. Например, пусть необходимо построить специальный метод, который принимает какой-то набор входных параметров. Такие параметры нужно использовать для создания нового типа данных, который будет применяться внутри области действия метода. Вдобавок желательно иметь возможность быстрого вывода данных с помощью метода `ToString()` и работы с другими членами `System.Object`. Всего сказанного можно достичь с помощью синтаксиса анонимных типов.

Определение анонимного типа

Анонимный тип определяется с использованием ключевого слова `var` (см. главу 3) в сочетании с синтаксисом инициализации объектов (см. главу 5). Ключевое слово `var` должно применяться из-за того, что компилятор будет автоматически генерировать новое определение класса на этапе компиляции (причем имя этого класса никогда не встретится в коде C#). Синтаксис инициализации применяется для сообщения компилятору о необходимости создания в новом типе закрытых поддерживающих полей и (допускающих только чтение) свойств.

В целях иллюстрации создайте новый проект консольного приложения по имени AnonymousTypes. Затем добавьте в класс Program показанный ниже метод, который формирует новый тип на лету, используя данные входных параметров:

```
static void BuildAnonymousType( string make, string color, int currSp )
{
  // Построить анонимный тип с применением входных аргументов.
  var car = new { Make = make, Color = color, Speed = currSp };

  // Обратите внимание, что теперь этот тип можно
  // использовать для получения данных свойств!
  Console.WriteLine("You have a {0} {1} going {2} MPH",
                    car.Color, car.Make, car.Speed);

  // Анонимные типы имеют специальные реализации каждого
  // виртуального метода System.Object. Например:
  Console.WriteLine("ToString() == {0}", car.ToString());
}
```

Обратите внимание, что помимо помещения кода внутрь функции анонимный тип можно также создавать непосредственно в строке:

```
Console.WriteLine("***** Fun with Anonymous Types *****\n");

// Создать анонимный тип, представляющий автомобиль.
var myCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };

// Вывести на консоль цвет и производителя.
Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Make);

// Вызвать вспомогательный метод для построения
// анонимного типа с указанием аргументов.
BuildAnonymousType("BMW", "Black", 90);

Console.ReadLine();
```

В настоящий момент достаточно понимать, что анонимные типы позволяют быстро моделировать "форму" данных с небольшими накладными расходами. Они являются лишь способом построения на лету нового типа данных, который поддерживает базовую инкапсуляцию через свойства и действует в соответствии с семантикой на основе значений. Чтобы уловить суть последнего утверждения, давайте посмотрим, каким образом компилятор C# строит анонимные типы на этапе компиляции, и в особенности — как он переопределяет члены System.Object.

Внутреннее представление анонимных типов

Все анонимные типы автоматически наследуются от System.Object и потому поддерживают все члены, предоставленные этим базовым классом. В результате можно вызывать метод ToString(), GetHashCode(), Equals() или GetType() на неявно типизированном объекте myCar. Предположим, что в классе Program определен следующий статический вспомогательный метод:

```
static void ReflectOverAnonymousType(object obj)
{
  Console.WriteLine("obj is an instance of: {0}",
    obj.GetType().Name);
  Console.WriteLine("Base class of {0} is {1}",
    obj.GetType().Name, obj.GetType().BaseType);
```

```
  Console.WriteLine("obj.ToString() == {0}",
    obj.ToString());
  Console.WriteLine("obj.GetHashCode() == {0}",
    obj.GetHashCode());
  Console.WriteLine();
}
```

Пусть вы вызвали метод `ReflectOverAnonymousType()`, передав ему объект `myCar` в качестве параметра:

```
Console.WriteLine("***** Fun with Anonymous Types *****\n");

// Создать анонимный тип, представляющий автомобиль.
var myCar = new {Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55};

// Выполнить рефлексию того, что сгенерировал компилятор.
ReflectOverAnonymousType(myCar);
...
Console.ReadLine();
```

Вывод будет выглядеть примерно так:

```
***** Fun with Anonymous Types *****

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() = { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() = -564053045
```

Первым делом обратите внимание в примере на то, что объект `myCar` имеет тип `<>f__AnonymousType0`3` (в вашем выводе имя типа может быть другим). Помните, что имя, назначаемое типу, полностью определяется компилятором и не доступно в коде C# напрямую.

Пожалуй, наиболее важно здесь то, что каждая пара "имя-значение", определенная с использованием синтаксиса инициализации объектов, отображается на идентично именованное свойство, доступное только для чтения, и соответствующее закрытое поддерживающее поле, которое допускает только инициализацию. Приведенный ниже код C# приблизительно отражает сгенерированный компилятором класс, применяемый для представления объекта `myCar` (его можно просмотреть посредством утилиты `ildasm.exe`):

```
private sealed class <>f__AnonymousType0'3'<'<Color>j__TPar',
  '<Make>j__TPar', <CurrentSpeed>j__TPar>'
  extends [System.Runtime][System.Object]
{
  // Поля только для инициализации.
  private initonly <Color>j__TPar <Color>i__Field;
  private initonly <CurrentSpeed>j__TPar <CurrentSpeed>i__Field;
  private initonly <Make>j__TPar <Make>i__Field;

  // Стандартный конструктор.
  public <>f__AnonymousType0(<Color>j__TPar Color,
    <Make>j__TPar Make, <CurrentSpeed>j__TPar CurrentSpeed);

  // Переопределенные методы.
  public override bool Equals(object value);
  public override int GetHashCode();
  public override string ToString();
```

```
  // Свойства только для чтения.
  <Color>j__TPar Color { get; }
  <CurrentSpeed>j__TPar CurrentSpeed { get; }
  <Make>j__TPar Make { get; }
}
```

Реализация методов ToString() и GetHashCode()

Все анонимные типы автоматически являются производными от `System.Object` и предоставляют переопределенные версии методов `Equals()`, `GetHashCode()` и `ToString()`. Реализация `ToString()` просто строит строку из пар "имя-значение". Вот пример:

```
public override string ToString()
{
  StringBuilder builder = new StringBuilder();
  builder.Append("{ Color = ");
  builder.Append(this.<Color>i__Field);
  builder.Append(", Make = ");
  builder.Append(this.<Make>i__Field);
  builder.Append(", CurrentSpeed = ");
  builder.Append(this.<CurrentSpeed>i__Field);
  builder.Append(" }");
  return builder.ToString();
}
```

Реализация `GetHashCode()` вычисляет хеш-значение, используя каждую переменную-член анонимного типа в качестве входных данных для типа `System.Collections.Generic.EqualityComparer<T>`. С такой реализацией `GetHashCode()` два анонимных типа будут выдавать одинаковые хеш-значения тогда (и только тогда), когда они обладают одним и тем же набором свойств, которым присвоены те же самые значения. Благодаря подобной реализации анонимные типы хорошо подходят для помещения внутрь контейнера `Hashtable`.

Семантика эквивалентности анонимных типов

Наряду с тем, что реализация переопределенных методов `ToString()` и `GetHashCode()` прямолинейна, вас может интересовать, как был реализован метод `Equals()`. Например, если определены две переменные "анонимных автомобилей" с одинаковыми наборами пар "имя-значение", то должны ли эти переменные считаться эквивалентными? Чтобы увидеть результат такого сравнения, дополните класс `Program` следующим новым методом:

```
static void EqualityTest()
{
  // Создать два анонимных класса с идентичными наборами
  // пар "имя-значение".
  var firstCar = new { Color = "Bright Pink", Make = "Saab",
                       CurrentSpeed = 55 };
  var secondCar = new { Color = "Bright Pink", Make = "Saab",
                        CurrentSpeed = 55 };

  // Считаются ли они эквивалентными, когда используется Equals()?
  if (firstCar.Equals(secondCar))
  {
```

```
    Console.WriteLine("Same anonymous object!");
                    // Тот же самый анонимный объект
  }
  else
  {
    Console.WriteLine("Not the same anonymous object!");
                    // Не тот же самый анонимный объект
  }
  // Можно ли проверить их эквивалентность с помощью операции ==?
  if (firstCar == secondCar)
  {
    Console.WriteLine("Same anonymous object!");
                    // Тот же самый анонимный объект
  }
  else
  {
    Console.WriteLine("Not the same anonymous object!");
                    // Не тот же самый анонимный объект
  }
  // Имеют ли эти объекты в основе один и тот же тип?
  if (firstCar.GetType().Name == secondCar.GetType().Name)
  {
    Console.WriteLine("We are both the same type!");
                    // Оба объекта имеют тот же самый тип
  }
  else
  {
    Console.WriteLine("We are different types!");
                    // Объекты относятся к разным типам
  }
  // Отобразить все детали.
  Console.WriteLine();
  ReflectOverAnonymousType(firstCar);
  ReflectOverAnonymousType(secondCar);
}
```

В результате вызова метода `EqualityTest()` получается несколько неожиданный вывод:

```
My car is a Bright Pink Saab.
You have a Black BMW going 90 MPH
ToString() == { Make = BMW, Color = Black, Speed = 90 }

Same anonymous object!
Not the same anonymous object!
We are both the same type!

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -925496951

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -925496951
```

Как видите, первая проверка, где вызывается `Equals()`, возвращает `true`, и потому на консоль выводится сообщение `Same anonymous object!` (Тот же самый анонимный объект). Причина в том, что сгенерированный компилятором метод `Equals()` при проверке эквивалентности применяет семантику на основе значений (т.е. проверяет значения каждого поля сравниваемых объектов).

Тем не менее, вторая проверка, в которой используется операция ==, приводит к выводу на консоль строки `Not the same anonymous object!` (Не тот же самый анонимный объект), что на первый взгляд выглядит несколько нелогично. Такой результат обусловлен тем, что анонимные типы *не* получают перегруженных версий операций проверки равенства (== и !=), поэтому при проверке эквивалентности объектов анонимных типов с применением операций равенства C# (вместо метода `Equals()`) проверяются *ссылки*, а не значения, поддерживаемые объектами.

Наконец, в финальной проверке (где исследуется имя лежащего в основе типа) обнаруживается, что объекты анонимных типов являются экземплярами одного и того же типа класса, сгенерированного компилятором (`<>f__AnonymousType0`3` в данном примере), т.к. `firstCar` и `secondCar` имеют одинаковые наборы свойств (`Color`, `Make` и `CurrentSpeed`).

Это иллюстрирует важный, но тонкий аспект: компилятор будет генерировать новое определение класса, только когда анонимный тип содержит *уникальные* имена свойств. Таким образом, если вы объявляете идентичные анонимные типы (в смысле имеющие те же самые имена свойств) внутри сборки, то компилятор генерирует единственное определение анонимного типа.

Анонимные типы, содержащие другие анонимные типы

Можно создавать анонимные типы, которые состоят из других анонимных типов. В качестве примера предположим, что требуется смоделировать заказ на приобретение, который хранит метку времени, цену и сведения о приобретаемом автомобиле. Вот новый (чуть более сложный) анонимный тип, представляющий такую сущность:

```
// Создать анонимный тип, состоящий из еще одного анонимного типа.
var purchaseItem = new {
  TimeBought = DateTime.Now,
  ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
  Price = 34.000};

ReflectOverAnonymousType(purchaseItem);
```

Сейчас вы уже должны понимать синтаксис, используемый для определения анонимных типов, но возможно все еще интересуетесь, где (и когда) применять такое языковое средство. Выражаясь кратко, объявления анонимных типов следует использовать умеренно, обычно только в случае применения набора технологий LINQ (см. главу 13). С учетом описанных ниже многочисленных ограничений анонимных типов вы никогда не должны отказываться от использования строго типизированных классов и структур просто из-за того, что это возможно.

- Контроль над именами анонимных типов отсутствует.
- Анонимные типы всегда расширяют `System.Object`.
- Поля и свойства анонимного типа всегда допускают только чтение.
- Анонимные типы не могут поддерживать события, специальные методы, специальные операции или специальные переопределения.

- Анонимные типы всегда неявно запечатаны.
- Экземпляры анонимных типов всегда создаются с применением стандартных конструкторов.

Однако при программировании с использованием набора технологий LINQ вы обнаружите, что во многих случаях такой синтаксис оказывается удобным, когда нужно быстро смоделировать общую *форму* сущности, а не ее функциональность.

Работа с типами указателей

Последняя тема главы касается средства C#, которое в подавляющем большинстве проектов .NET Core применяется реже всех остальных.

На заметку! В последующих примерах предполагается наличие у вас определенных навыков манипулирования указателями в языке C++. Если это не так, тогда можете спокойно пропустить данную тему. В большинстве приложений C# указатели не используются.

В главе 4 вы узнали, что в рамках платформы .NET Core определены две крупные категории данных: типы значений и ссылочные типы. По правде говоря, на самом деле есть еще и третья категория: *типы указателей*. Для работы с типами указателей доступны специфичные операции и ключевые слова (табл. 11.2), которые позволяют обойти схему управления памятью исполняющей среды .NET 5 и взять дело в свои руки.

Таблица 11.2. Операции и ключевые слова C#, связанные с указателями

Операция/ ключевое слово	Назначение
*	Эта операция применяется для создания переменной указателя (т.е. переменной, которая представляет непосредственное местоположение в памяти). Как и в языке C++, та же самая операция используется для разыменования указателя
&	Эта операция применяется для получения адреса переменной в памяти
->	Эта операция используется для доступа к полям типа, представленного указателем (небезопасная версия операции точки в C#)
[]	Эта операция (в небезопасном контексте) позволяет индексировать область памяти, на которую указывает переменная указателя (если вы программировали на языке C++, то вспомните о взаимодействии между переменной указателя и операцией [])
++, --	В небезопасном контексте операции инкремента и декремента могут применяться к типам указателей
+, -	В небезопасном контексте операции сложения и вычитания могут применяться к типам указателей
==, !=, <, >, <=, >=	В небезопасном контексте операции сравнения и эквивалентности могут применяться к типам указателей
stackalloc	В небезопасном контексте ключевое слово stackalloc может использоваться для размещения массивов C# прямо в стеке
fixed	В небезопасном контексте ключевое слово fixed может применяться для временного закрепления переменной, чтобы впоследствии удалось найти ее адрес

Перед погружением в детали следует еще раз подчеркнуть, что вам очень *редко, если вообще когда-нибудь*, понадобится использовать типы указателей. Хотя C# позволяет опуститься на уровень манипуляций указателями, помните, что исполняющая среда .NET Core не имеет абсолютно никакого понятия о ваших намерениях. Соответственно, если вы неправильно управляете указателем, то сами и будете отвечать за последствия. С учетом этих предупреждений возникает вопрос: когда в принципе может возникнуть необходимость работы с типами указателей? Существуют две распространенные ситуации.

- Нужно оптимизировать избранные части приложения, напрямую манипулируя памятью за рамками ее управления со стороны исполняющей среды .NET 5.
- Необходимо вызывать методы из DLL-библиотеки, написанной на C, либо из сервера COM, которые требуют передачи типов указателей в качестве параметров. Но даже в таком случае часто можно обойтись без применения типов указателей, отдав предпочтение типу `System.IntPtr` и членам типа `System.Runtime.InteropServices.Marshal`.

Если вы решили задействовать данное средство языка C#, тогда придется информировать компилятор C# о своих намерениях, разрешив проекту поддерживать "небезопасный код". Создайте новый проект консольного приложения по имени `UnsafeCode` и включите поддержку небезопасного кода, добавив в файл `UnsafeCode.csproj` следующие строки:

```
<PropertyGroup>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

Для установки этого свойства в Visual Studio предлагается графический пользовательский интерфейс. Откройте окно свойств проекта. В раскрывающемся списке Configuration (Конфигурация) выберите вариант All Configurations (Все конфигурации), перейдите на вкладку Build (Сборка) и отметьте флажок Allow unsafe code (Разрешить небезопасный код), как показано на рис. 11.1.

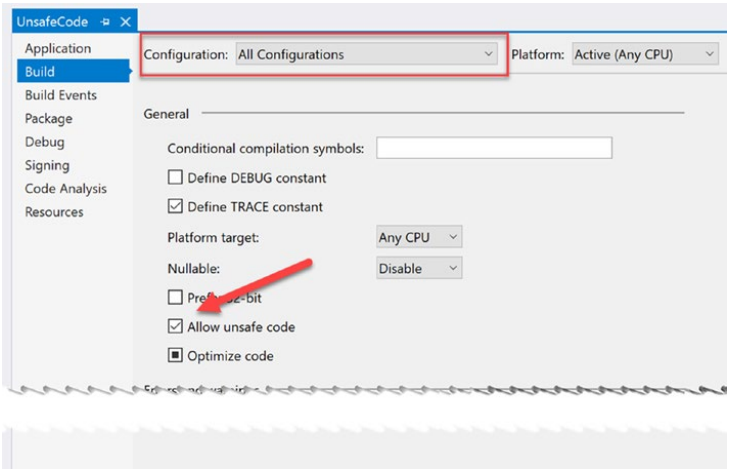

Рис. 11.1. Включение поддержки небезопасного кода в Visual Studio

Ключевое слово unsafe

Для работы с указателями в C# должен быть специально объявлен блок "небезопасного кода" с использованием ключевого слова unsafe (любой код, который не помечен ключевым словом unsafe, автоматически считается "безопасным"). Например, в следующем файле Program.cs объявляется область небезопасного кода внутри операторов верхнего уровня:

```
using System;
using UnsafeCode;

Console.WriteLine("***** Calling method with unsafe code *****");

unsafe
{
  // Здесь работаем с указателями!
}
// Здесь работа с указателями невозможна!
```

В дополнение к объявлению области небезопасного кода внутри метода можно строить "небезопасные" структуры, классы, члены типов и параметры. Ниже приведено несколько примеров (типы Node и Node2 в текущем проекте определять не нужно):

```
// Эта структура целиком является небезопасной и может
// использоваться только в небезопасном контексте.
unsafe struct Node
{
  public int Value;
  public Node* Left;
  public Node* Right;
}

// Эта структура безопасна, но члены Node2* - нет.
// Формально извне небезопасного контекста можно
// обращаться к Value, но не к Left и Right.
public struct Node2
{
  public int Value;

  // Эти члены доступны только в небезопасном контексте!
  public unsafe Node2* Left;
  public unsafe Node2* Right;
}
```

Методы (статические либо уровня экземпляра) также могут быть помечены как небезопасные. Предположим, что какой-то статический метод будет использовать логику указателей. Чтобы обеспечить возможность вызова данного метода только из небезопасного контекста, его можно определить так:

```
static unsafe void SquareIntPointer(int* myIntPointer)
{
  // Возвести значение в квадрат просто для тестирования.
  *myIntPointer *= *myIntPointer;
}
```

Конфигурация метода требует, чтобы вызывающий код обращался к методу SquareIntPointer() следующим образом:

```
unsafe
{
  int myInt = 10;
  // Нормально, мы находимся в небезопасном контексте.
  SquareIntPointer(&myInt);
  Console.WriteLine("myInt: {0}", myInt);
}
int myInt2 = 5;
// Ошибка на этапе компиляции!
// Это должно делаться в небезопасном контексте!
SquareIntPointer(&myInt2);
Console.WriteLine("myInt: {0}", myInt2);
```

Если вы не хотите вынуждать вызывающий код помещать такой вызов внутрь небезопасного контекста, то можете поместить все операторы верхнего уровня в блок `unsafe`. При использовании в качестве точки входа метода `Main()` можете пометить `Main()` ключевым словом `unsafe`. В таком случае приведенный ниже код скомпилируется:

```
static unsafe void Main(string[] args)
{
  int myInt2 = 5;
  SquareIntPointer(&myInt2);
  Console.WriteLine("myInt: {0}", myInt2);
}
```

Запустив такую версию кода, вы получите следующий вывод:

```
myInt: 25
```

На заметку! Важно отметить, что термин "небезопасный" был выбран небезосновательно. Прямой доступ к стеку и работа с указателями может приводить к неожиданным проблемам с вашим приложением, а также с компьютером, на котором оно функционирует. Если вам приходится иметь дело с небезопасным кодом, тогда будьте крайне внимательны.

Работа с операциями * и &

После установления небезопасного контекста можно строить указатели и типы данных с помощью операции `*`, а также получать адрес указываемых данных посредством операции `&`. В отличие от C или C++ в языке C# операция `*` применяется только к лежащему в основе типу, а не является префиксом имени каждой переменной указателя. Например, взгляните на показанный далее код, демонстрирующий правильный и неправильный способы объявления указателей на целочисленные переменные:

```
// Нет! В C# это некорректно!
int *pi, *pj;
// Да! Так поступают в C#.
int* pi, pj;
```

Рассмотрим следующий небезопасный метод:

```
static unsafe void PrintValueAndAddress()
{
  int myInt;
```

```
  // Определить указатель на int и присвоить ему адрес myInt.
  int* ptrToMyInt = &myInt;

  // Присвоить значение myInt, используя обращение через указатель.
  *ptrToMyInt = 123;

  // Вывести некоторые значения.
  Console.WriteLine("Value of myInt {0}", myInt);
                    // значение myInt
  Console.WriteLine("Address of myInt {0:X}", (int)&ptrToMyInt);
                    // адрес myInt
}
```

В результате запуска этого метода из блока `unsafe` вы получите такой вывод:

```
**** Print Value And Address ****
Value of myInt 123
Address of myInt 90F7E698
```

Небезопасная (и безопасная) функция обмена

Разумеется, объявлять указатели на локальные переменные, чтобы просто присваивать им значения (как в предыдущем примере), никогда не понадобится и к тому же неудобно. В качестве более практичного примера небезопасного кода предположим, что необходимо построить функцию обмена с использованием арифметики указателей:

```
unsafe static void UnsafeSwap(int* i, int* j)
{
  int temp = *i;
  *i = *j;
  *j = temp;
}
```

Очень похоже на язык C, не так ли? Тем не менее, учитывая предшествующую работу, вы должны знать, что можно было бы написать безопасную версию алгоритма обмена с применением ключевого слова `ref` языка C#:

```
static void SafeSwap(ref int i, ref int j)
{
  int temp = i;
  i = j;
  j = temp;
}
```

Функциональность обеих версий метода идентична, доказывая тем самым, что прямые манипуляции указателями в C# не являются обязательными. Ниже показана логика вызова, использующая безопасные операторы верхнего уровня, но с небезопасным контекстом:

```
Console.WriteLine("***** Calling method with unsafe code *****");
// Значения, подлежащие обмену.
int i = 10, j = 20;

// "Безопасный" обмен значений местами.
Console.WriteLine("\n***** Safe swap *****");
Console.WriteLine("Values before safe swap: i = {0}, j = {1}", i, j);
SafeSwap(ref i, ref j);
Console.WriteLine("Values after safe swap: i = {0}, j = {1}", i, j);
```

```
// "Небезопасный" обмен значений местами.
Console.WriteLine("\n***** Unsafe swap *****");
Console.WriteLine("Values before unsafe swap: i = {0}, j = {1}", i, j);
unsafe { UnsafeSwap(&i, &j); }
Console.WriteLine("Values after unsafe swap: i = {0}, j = {1}", i, j);
Console.ReadLine();
```

Доступ к полям через указатели (операция ->)

Теперь предположим, что определена простая безопасная структура `Point`:

```
struct Point
{
  public int x;
  public int y;
  public override string ToString() => $"({x}, {y})";
}
```

В случае объявления указателя на тип `Point` для доступа к открытым членам структуры понадобится применять операцию доступа к полям (имеющую вид ->). Как упоминалось в табл. 11.2, она представляет собой небезопасную версию стандартной (безопасной) операции точки (.). В сущности, используя операцию обращения к указателю (*), можно разыменовывать указатель для применения операции точки. Взгляните на следующий небезопасный метод:

```
static unsafe void UsePointerToPoint()
{
  // Доступ к членам через указатель.
  Point point;
  Point* p = &point;
  p->x = 100;
  p->y = 200;
  Console.WriteLine(p->ToString());
  // Доступ к членам через разыменованный указатель.
  Point point2;
  Point* p2 = &point2;
  (*p2).x = 100;
  (*p2).y = 200;
  Console.WriteLine((*p2).ToString());
}
```

Ключевое слово stackalloc

В небезопасном контексте может возникнуть необходимость в объявлении локальной переменной, для которой память выделяется непосредственно в стеке вызовов (и потому она не обрабатывается сборщиком мусора .NET Core). Для этого в языке C# предусмотрено ключевое слово `stackalloc`, которое является эквивалентом функции `_alloca` библиотеки времени выполнения C. Вот простой пример:

```
static unsafe string UnsafeStackAlloc()
{
  char* p = stackalloc char[52];
  for (int k = 0; k < 52; k++)
  {
    p[k] = (char)(k + 65)k;
  }
  return new string(p);
}
```

Закрепление типа посредством ключевого слова `fixed`

В предыдущем примере вы видели, что выделение фрагмента памяти внутри небезопасного контекста может делаться с помощью ключевого слова `stackalloc`. Из-за природы операции `stackalloc` выделенная память очищается, как только выделяющий ее метод возвращает управление (т.к. память распределена в стеке). Однако рассмотрим более сложный пример. Во время исследования операции -> создавался тип значения по имени `Point`. Как и все типы значений, выделяемая его экземплярам память исчезает из стека по окончании выполнения. Предположим, что тип `Point` взамен определен как *ссылочный*:

```
class PointRef  // <= Переименован и обновлен.
{
  public int x;
  public int y;
  public override string ToString() => $"({x}, {y})";
}
```

Как вам известно, если в вызывающем коде объявляется переменная типа `Point`, то память для нее выделяется в куче, поддерживающей сборку мусора. И тут возникает животрепещущий вопрос: а что если небезопасный контекст пожелает взаимодействовать с этим объектом (или любым другим объектом из кучи)? Учитывая, что сборка мусора может произойти в любое время, вы только вообразите, какие проблемы возникнут при обращении к членам `Point` именно в тот момент, когда происходит реорганизация кучи! Теоретически может случиться так, что небезопасный контекст попытается взаимодействовать с членом, который больше не доступен или был перемещен в другое место кучи после ее очистки с учетом поколений (что является очевидной проблемой).

Для фиксации переменной ссылочного типа в памяти из небезопасного контекста язык C# предлагает ключевое слово `fixed`. Оператор `fixed` устанавливает указатель на управляемый тип и "закрепляет" такую переменную на время выполнения кода. Без `fixed` от указателей на управляемые переменные было бы мало толку, поскольку сборщик мусора может перемещать переменные в памяти непредсказуемым образом. (На самом деле компилятор C# даже не позволит установить указатель на управляемую переменную, если оператор `fixed` отсутствует.)

Таким образом, если вы создали объект `Point` и хотите взаимодействовать с его членами, тогда должны написать следующий код (либо иначе получить ошибку на этапе компиляции):

```
unsafe static void UseAndPinPoint()
{
  PointRef pt = new PointRef
  {
    x = 5,
    y = 6
  };
  // Закрепить указатель pt на месте, чтобы он не мог
  // быть перемещен или уничтожен сборщиком мусора.
  fixed (int* p = &pt.x)
  {
    // Использовать здесь переменную int*!
  }
```

```
  // Указатель pt теперь не закреплен и готов
  // к сборке мусора после завершения метода.
  Console.WriteLine ("Point is: {0}", pt);
}
```

Выражаясь кратко, ключевое слово `fixed` позволяет строить оператор, который фиксирует ссылочную переменную в памяти, чтобы ее адрес оставался постоянным на протяжении работы оператора (или блока операторов). Каждый раз, когда вы взаимодействуете со ссылочным типом из контекста небезопасного кода, закрепление ссылки обязательно.

Ключевое слово `sizeof`

Последнее ключевое слово C#, связанное с небезопасным кодом — `sizeof`. Как и в C++, ключевое слово `sizeof` в C# используется для получения размера в байтах *встроенного типа данных*, но не специального типа, разве только в небезопасном контексте. Например, показанный ниже метод не нуждается в объявлении "небезопасным", т.к. все аргументы ключевого слова `sizeof` относятся к встроенным типам:

```
static void UseSizeOfOperator()
{
  Console.WriteLine("The size of short is {0}.", sizeof(short));
                    // размер short
  Console.WriteLine("The size of int is {0}.", sizeof(int));
                    // размер int
  Console.WriteLine("The size of long is {0}.", sizeof(long));
                    // размер long
}
```

Тем не менее, если вы хотите получить размер специальной структуры `Point`, то метод `UseSizeOfOperator()` придется модифицировать (обратите внимание на добавление ключевого слова `unsafe`):

```
unsafe static void UseSizeOfOperator()
{
  ...
  unsafe {
    Console.WriteLine("The size of Point is {0}.", sizeof(Point));
                      // размер Point
  }
}
```

Итак, обзор нескольких более сложных средств языка программирования C# завершен. Напоследок снова необходимо отметить, что в большинстве проектов .NET эти средства могут вообще не понадобиться (особенно указатели). Тем не менее, как будет показано в последующих главах, некоторые средства действительно полезны (и даже обязательны) при работе с API-интерфейсами LINQ, в частности расширяющие методы и анонимные типы.

Резюме

Целью главы было углубление знаний языка программирования C#. Первым делом мы исследовали разнообразные более сложные конструкции в типах (индексаторные методы, перегруженные операции и специальные процедуры преобразования).

Затем мы рассмотрели роль расширяющих методов и анонимных типов. Как вы увидите в главе 13, эти средства удобны при работе с API-интерфейсами LINQ (хотя при желании их можно применять в коде повсеместно). Вспомните, что анонимные методы позволяют быстро моделировать "форму" типа, в то время как расширяющие методы дают возможность добавлять новую функциональность к типам без необходимости в определении подклассов.

Финальная часть главы была посвящена небольшому набору менее известных ключевых слов (`sizeof`, `unsafe` и т.п.); наряду с ними рассматривалась работа с низкоуровневыми типами указателей. Как было установлено в процессе исследования типов указателей, в большинстве приложений C# их *никогда* не придется использовать.

ГЛАВА 12

Делегаты, события и лямбда-выражения

Вплоть до настоящего момента в большинстве разработанных приложений к операторам верхнего уровня внутри файла `Program.cs` добавлялись разнообразные порции кода, тем или иным способом отправляющие запросы к заданному объекту. Однако многие приложения требуют, чтобы объект имел возможность обращаться *обратно* к сущности, которая его создала, используя механизм обратного вызова. Хотя механизмы обратного вызова могут применяться в любом приложении, они особенно важны в графических пользовательских интерфейсах, где элементы управления (такие как кнопки) нуждаются в вызове внешних методов при надлежащих обстоятельствах (когда произведен щелчок на кнопке, курсор мыши наведен на поверхность кнопки и т.д.).

В рамках платформы .NET Core предпочтительным средством определения и реагирования на обратные вызовы в приложении является тип *делегата*. По существу тип делегата .NET Core — это безопасный в отношении типов объект, "указывающий" на метод или список методов, которые могут быть вызваны позднее. Тем не менее, в отличие от традиционного указателя на функцию C++ делегаты представляют собой классы, которые обладают встроенной поддержкой группового и асинхронного вызова методов.

На заметку! В предшествующих версиях .NET делегаты обеспечивали вызов асинхронных методов с помощью `BeginInvoke()/EndInvoke()`. Хотя компилятор по-прежнему генерирует методы `BeginInvoke()/EndInvoke()`, в .NET Core они не поддерживаются. Причина в том, что шаблон с `IAsyncResult` и `BeginInvoke()/BeginInvoke()`, используемый делегатами, был заменен асинхронным шаблоном на основе задач. Асинхронное выполнение подробно обсуждается в главе 15.

В текущей главе вы узнаете, каким образом создавать и манипулировать типами делегатов, а также использовать ключевое слово `event` языка C#, которое облегчает работу с типами делегатов. По ходу дела вы также изучите несколько языковых средств C#, ориентированных на делегаты и события, в том числе анонимные методы и групповые преобразования методов.

Глава завершается исследованием *лямбда-выражений*. С помощью лямбда-операции C# (=>) можно указывать блок операторов кода (и подлежащие передаче им параметры) везде, где требуется строго типизированный делегат. Как будет показано, лямбда-выражение — не более чем замаскированный анонимный метод и является упрощенным подходом к работе с делегатами. Вдобавок та же самая операция (в .NET Framework 4.6

и последующих версиях) может применяться для реализации метода или свойства, содержащего единственный оператор, посредством лаконичного синтаксиса.

Понятие типа делегата

Прежде чем формально определить делегаты, давайте ненадолго оглянемся назад. Исторически сложилось так, что в API-интерфейсе Windows часто использовались указатели на функции в стиле C для создания сущностей под названием *функции обратного вызова* или просто *обратные вызовы*. С помощью обратных вызовов программисты могли конфигурировать одну функцию так, чтобы она обращалась к другой функции в приложении (т.е. делала обратный вызов). С применением такого подхода разработчики Windows-приложений имели возможность обрабатывать щелчки на кнопках, перемещение курсора мыши, выбор пунктов меню и общие двусторонние коммуникации между двумя сущностями в памяти.

В .NET и .NET Core обратные вызовы выполняются в безопасной в отношении типов объектно-ориентированной манере с использованием *делегатов*. Делегат — это безопасный в отношении типов объект, указывающий на другой метод или возможно на список методов приложения, которые могут быть вызваны в более позднее время. В частности, делегат поддерживает три важных порции информации:

- *адрес* метода, вызовы которого он делает;
- *аргументы* (если есть) вызываемого метода;
- *возвращаемое значение* (если есть) вызываемого метода.

На заметку! Делегаты .NET Core могут указывать либо на статические методы, либо на методы экземпляра.

После того как делегат создан и снабжен необходимой информацией, он может во время выполнения динамически вызывать метод или методы, на которые указывает.

Определение типа делегата в C#

Для определения типа делегата в языке C# применяется ключевое слово `delegate`. Имя типа делегата может быть любым желаемым. Однако сигнатура определяемого делегата должна совпадать с сигнатурой метода или методов, на которые он будет указывать. Например, приведенный ниже тип делегата (по имени `BinaryOp`) может указывать на любой метод, который возвращает целое число и принимает два целых числа в качестве входных параметров (позже в главе вы самостоятельно построите такой делегат, а пока он представлен лишь кратко):

```
// Этот делегат может указывать на любой метод, который принимает
// два целочисленных значения и возвращает целочисленное значение.
public delegate int BinaryOp(int x, int y);
```

Когда компилятор C# обрабатывает тип делегата, он автоматически генерирует запечатанный (`sealed`) класс, производный от `System.MulticastDelegate`. Этот класс (в сочетании со своим базовым классом `System.Delegate`) предоставляет необходимую инфраструктуру для делегата, которая позволяет хранить список методов, подлежащих вызову в будущем. Например, если вы изучите делегат `BinaryOp` с помощью утилиты `ildasm.exe`, то обнаружите показанные ниже детали (вскоре вы построите полный пример):

```
//    ------------------------------------------------------
//    TypDefName: SimpleDelegate.BinaryOp
//    Extends   : System.MulticastDelegate
//    Method #1
//    ------------------------------------------------------
//          MethodName: .ctor
//          ReturnType: Void
//          2 Arguments
//                Argument #1: Object
//                Argument #2: I
//
//    Method #2
//    ------------------------------------------------------
//          MethodName: Invoke
//          ReturnType: I4
//          2 Arguments
//                Argument #1: I4
//                Argument #2: I4
//          2 Parameters
//                (1) ParamToken : Name : x flags: [none]
//                (2) ParamToken : Name : y flags: [none]
//
//    Method #3
//    ------------------------------------------------------
//          MethodName: BeginInvoke
//          ReturnType: Class System.IAsyncResult
//          4 Arguments
//                Argument #1: I4
//                Argument #2: I4
//                Argument #3: Class System.AsyncCallback
//                Argument #4: Object
//          4 Parameters
//                (1) ParamToken : Name : x flags: [none]
//                (2) ParamToken : Name : y flags: [none]
//                (3) ParamToken : Name : callback flags: [none]
//                (4) ParamToken : Name : object flags: [none]
//
//    Method #4
//    ------------------------------------------------------
//          MethodName: EndInvoke
//          ReturnType: I4 (int32)
//          1 Arguments
//                Argument #1: Class System.IAsyncResult
//          1 Parameters
//                (1) ParamToken : Name : result flags: [none]
```

Как видите, в сгенерированном компилятором классе `BinaryOp` определены три открытых метода. Главным методом в .NET Core является `Invoke()`, т.к. он используется для вызова каждого метода, поддерживаемого объектом делегата, в *синхронной* манере; это означает, что вызывающий код должен ожидать завершения вызова, прежде чем продолжить свою работу. Довольно странно, но синхронный метод `Invoke()` может не нуждаться в явном вызове внутри вашего кода C#. Вскоре будет показано, что `Invoke()` вызывается "за кулисами", когда вы применяете соответствующий синтаксис C#.

На заметку! Несмотря на то что методы `BeginInvoke()` и `EndInvoke()` генерируются, они не поддерживаются при запуске вашего кода под управлением .NET Core. Это может разочаровывать, поскольку в случае их использования вы получите ошибку не на этапе компиляции, а во время выполнения.

Так благодаря чему же компилятор знает, как определять метод `Invoke()`? Для понимания процесса ниже приведен код сгенерированного компилятором класса `BinaryOp` (полужирным курсивом выделены элементы, указанные в определении типа делегата):

```
sealed class BinaryOp : System.MulticastDelegate
{
  public int Invoke(int x, int y);
  ...
}
```

Первым делом обратите внимание, что параметры и возвращаемый тип для метода `Invoke()` в точности соответствуют определению делегата `BinaryOp`.

Давайте рассмотрим еще один пример. Предположим, что определен тип делегата, который может указывать на любой метод, возвращающий значение `string` и принимающий три входных параметра типа `System.Boolean`:

```
public delegate string MyDelegate(bool a, bool b, bool c);
```

На этот раз сгенерированный компилятором класс можно представить так:

```
sealed class MyDelegate : System.MulticastDelegate
{
  public string Invoke(bool a, bool b, bool c);
  ...
}
```

Делегаты могут также "указывать" на методы, которые содержат любое количество параметров `out` и `ref` (а также параметры типа массивов, помеченные с помощью ключевого слова `params`). Пусть имеется следующий тип делегата:

```
public delegate string MyOtherDelegate(out bool a, ref bool b, int c);
```

Сигнатура метода `Invoke()` выглядит вполне ожидаемо.

Подводя итоги, отметим, что определение типа делегата C# дает в результате запечатанный класс со сгенерированным компилятором методом, в котором типы параметров и возвращаемые типы основаны на объявлении делегата. Базовый шаблон может быть приближенно описан с помощью следующего псевдокода:

```
// Это лишь псевдокод!
public sealed class ИмяДелегата : System.MulticastDelegate
{
  public возвращаемоеЗначениеДелегата
     Invoke(всеВходныеRefиOutПараметрыДелегата);
}
```

Базовые классы System.MulticastDelegate и System.Delegate

Итак, когда вы строите тип с применением ключевого слова delegate, то неявно объявляете тип класса, производного от System.MulticastDelegate. Данный класс предоставляет своим наследникам доступ к списку, который содержит адреса методов, поддерживаемых типом делегата, а также несколько дополнительных методов (и перегруженных операций) для взаимодействия со списком вызовов. Ниже приведены избранные методы класса System.MulticastDelegate:

```
public abstract class MulticastDelegate : Delegate
{
  // Возвращает список методов, на которые "указывает" делегат.
  public sealed override Delegate[] GetInvocationList();

  // Перегруженные операции.
  public static bool operator ==(MulticastDelegate d1,
                                 MulticastDelegate d2);
  public static bool operator !=(MulticastDelegate d1,
                                 MulticastDelegate d2);

  // Используются внутренне для управления списком методов,
  // поддерживаемых делегатом.
  private IntPtr _invocationCount;
  private object _invocationList;
}
```

Класс System.MulticastDelegate получает дополнительную функциональность от своего родительского класса System.Delegate. Вот фрагмент его определения:

```
public abstract class Delegate : ICloneable, ISerializable
{
  // Методы для взаимодействия со списком функций.
  public static Delegate Combine(params Delegate[] delegates);
  public static Delegate Combine(Delegate a, Delegate b);
  public static Delegate Remove(Delegate source, Delegate value);
  public static Delegate RemoveAll(Delegate source, Delegate value);

  // Перегруженные операции.
  public static bool operator ==(Delegate d1, Delegate d2);
  public static bool operator !=(Delegate d1, Delegate d2);

  // Свойства, открывающие доступ к цели делегата.
  public MethodInfo Method { get; }
  public object Target { get; }
}
```

Имейте в виду, что вы никогда не сможете напрямую наследовать от таких базовых классов в своем коде (попытка наследования приводит к ошибке на этапе компиляции). Тем не менее, когда вы используете ключевое слово delegate, то тем самым неявно создаете класс, который "является" MulticastDelegate. В табл. 12.1 описаны основные члены, общие для всех типов делегатов.

Таблица 12.1. Избранные члены классов `System.MulticastDelegate/System.Delegate`

Член	Назначение
`Method`	Это свойство возвращает объект `System.Reflection.Method`, который представляет детали статического метода, поддерживаемого делегатом
`Target`	Если подлежащий вызову метод определен на уровне объектов (т.е. он нестатический), тогда это свойство возвращает объект, который представляет метод, поддерживаемый делегатом. Если возвращенное `Target` значение равно `null`, то подлежащий вызову метод является статическим
`Combine()`	Этот статический метод добавляет метод в список, поддерживаемый делегатом. В языке C# данный метод вызывается с применением перегруженной операции += в качестве сокращенной записи
`GetInvokationList()`	Этот метод возвращает массив объектов `System.Delegate`, каждый из которых представляет определенный метод, доступный для вызова
`Remove()` `RemoveAll()`	Эти статические методы удаляют метод (или все методы) из списка вызовов делегата. В языке C# метод `Remove()` может быть вызван косвенно с использованием перегруженной операции -=

Пример простейшего делегата

На первый взгляд делегаты могут показаться несколько запутанными. Рассмотрим для начала простой проект консольного приложения (по имени `SimpleDelegate`), в котором применяется определенный ранее тип делегата `BinaryOp`. Ниже показан полный код с последующим анализом:

```
// SimpleMath.cs
namespace SimpleDelegate
{
  // Этот класс содержит методы, на которые
  // будет указывать BinaryOp.
  public class SimpleMath
  {
    public static int Add(int x, int y) => x + y;
    public static int Subtract(int x, int y) => x - y;
  }
}

// Program.cs
using System;
using SimpleDelegate;

Console.WriteLine("***** Simple Delegate Example *****\n");

// Создать объект делегата BinaryOp, который
// "указывает" на SimpleMath.Add().
BinaryOp b = new BinaryOp(SimpleMath.Add);

// Вызвать метод Add() косвенно с использованием объекта делегата.
Console.WriteLine("10 + 10 is {0}", b(10, 10));
Console.ReadLine();
```

```
// Дополнительные определения типов должны находиться
// в конце операторов верхнего уровня.
// Этот делегат может указывать на любой метод,
// принимающий два целых числа и возвращающий целое число.
public delegate int BinaryOp(int x, int y);
```

На заметку! Вспомните из главы 3, что дополнительные определения типов (делегат `BinaryOp` в этом примере) должны располагаться после всех операторов верхнего уровня.

И снова обратите внимание на формат объявления типа делегата `BinaryOp`; он определяет, что объекты делегата `BinaryOp` могут указывать на любой метод, принимающий два целочисленных значения и возвращающий целочисленное значение (действительное имя метода, на который он указывает, к делу не относится). Здесь мы создали класс по имени `SimpleMath`, определяющий два статических метода, которые соответствуют шаблону, определяемому делегатом `BinaryOp`.

Когда вы хотите присвоить целевой метод заданному объекту делегата, просто передайте имя нужного метода конструктору делегата:

```
// Создать объект делегата BinaryOp, который
// "указывает" на SimpleMath.Add().
BinaryOp b = new BinaryOp(SimpleMath.Add);
```

На данной стадии метод, на который указывает делегат, можно вызывать с использованием синтаксиса, выглядящего подобным прямому вызову функции:

```
// На самом деле здесь вызывается метод Invoke()!
Console.WriteLine("10 + 10 is {0}", b(10, 10));
```

"За кулисами" исполняющая среда вызывает сгенерированный компилятором метод `Invoke()` на вашем производном от `MulticastDelegate` классе. В этом можно удостовериться, открыв сборку в утилите `ildasm.exe` и просмотрев код CIL внутри метода `Main()`:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  ...
  callvirt   instance int32 BinaryOp::Invoke(int32, int32)
}
```

Язык C# вовсе не требует явного вызова метода `Invoke()` внутри вашего кода. Поскольку `BinaryOp` может указывать на методы, которые принимают два аргумента, следующий оператор тоже допустим:

```
Console.WriteLine("10 + 10 is {0}", b.Invoke(10, 10));
```

Вспомните, что делегаты .NET Core *безопасны в отношении типов*. Следовательно, если вы попытаетесь передать делегату метод, который не соответствует его шаблону, то получите ошибку на этапе компиляции. В целях иллюстрации предположим, что в классе `SimpleMath` теперь определен дополнительный метод по имени `SquareNumber()`, принимающий единственный целочисленный аргумент:

```
public class SimpleMath
{
  public static int SquareNumber(int a) => a * a;
}
```

Учитывая, что делегат BinaryOp может указывать *только* на методы, которые принимают два целочисленных значения и возвращают целочисленное значение, представленный ниже код некорректен и приведет к ошибке на этапе компиляции:

```
// Ошибка на этапе компиляции! Метод не соответствует шаблону делегата!
BinaryOp b2 = new BinaryOp(SimpleMath.SquareNumber);
```

Исследование объекта делегата

Давайте усложним текущий пример, создав в классе Program статический метод (по имени DisplayDelegateInfo()). Он будет выводить на консоль имена методов, поддерживаемых объектом делегата, а также имя класса, определяющего метод. Для этого организуется итерация по массиву System.Delegate, возвращенному методом GetInvocationList(), с обращением к свойствам Target и Method каждого объекта:

```
static void DisplayDelegateInfo(Delegate delObj)
{
  // Вывести имена всех членов в списке вызовов делегата.
  foreach (Delegate d in delObj.GetInvocationList())
  {
    Console.WriteLine("Method Name: {0}", d.Method);  // имя метода
    Console.WriteLine("Type Name: {0}", d.Target);    // имя типа
  }
}
```

Предполагая, что в метод Main() добавлен вызов нового вспомогательного метода:

```
BinaryOp b = new BinaryOp(SimpleMath.Add);
DisplayDelegateInfo(b);
```

вывод приложения будет таким:

```
***** Simple Delegate Example *****

Method Name: Int32 Add(Int32, Int32)
Type Name:
10 + 10 is 20
```

Обратите внимание, что при обращении к свойству Target имя целевого класса (SimpleMath) в настоящий момент *не* отображается. Причина в том, что делегат BinaryOp указывает на *статический метод*, и потому объект для ссылки попросту отсутствует! Однако если сделать методы Add() и Substract() нестатическими (удалив ключевое слово static из их объявлений), тогда можно будет создавать экземпляр класса SimpleMath и указывать методы для вызова с применением ссылки на объект:

```
using System;
using SimpleDelegate;
Console.WriteLine("***** Simple Delegate Example *****\n");

// Делегаты могут также указывать на методы экземпляра.
SimpleMath m = new SimpleMath();
BinaryOp b = new BinaryOp(m.Add);

// Вывести сведения об объекте.
DisplayDelegateInfo(b);
Console.WriteLine("10 + 10 is {0}", b(10, 10));
Console.ReadLine();
```

В данном случае вывод будет выглядеть следующим образом:

```
***** Simple Delegate Example *****

Method Name: Int32 Add(Int32, Int32)
Type Name: SimpleDelegate.SimpleMath
10 + 10 is 20
```

Отправка уведомлений о состоянии объекта с использованием делегатов

Очевидно, что предыдущий пример `SimpleDelegate` был чисто иллюстративным по своей природе, т.к. нет особых причин создавать делегат просто для того, чтобы сложить два числа. Рассмотрим более реалистичный пример, в котором делегаты применяются для определения класса `Car`, обладающего способностью информировать внешние сущности о текущем состоянии двигателя. В таком случае нужно выполнить перечисленные ниже действия.

1. Определить новый тип делегата, который будет использоваться для отправки уведомлений вызывающему коду.
2. Объявить переменную-член этого типа делегата в классе `Car`.
3. Создать в классе `Car` вспомогательную функцию, которая позволяет вызывающему коду указывать метод для обратного вызова.
4. Реализовать метод `Accelerate()` для обращения к списку вызовов делегата в подходящих обстоятельствах.

Для начала создайте новый проект консольного приложения по имени `CarDelegate`. Определите в нем новый класс `Car`, начальный код которого показан ниже:

```
using System;
using System.Linq;

namespace CarDelegate
{
  public class Car
  {
    // Внутренние данные состояния.
    public int CurrentSpeed { get; set; }
    public int MaxSpeed { get; set; } = 100;
    public string PetName { get; set; }

    // Исправен ли автомобиль?
    private bool _carIsDead;

    // Конструкторы класса.
    public Car() {}
    public Car(string name, int maxSp, int currSp)
    {
      CurrentSpeed = currSp;
      MaxSpeed = maxSp;
      PetName = name;
    }
  }
}
```

А теперь модифицируйте его, выполнив первые три действия из числа указанных выше:

```
public class Car
{
  ...
  // 1. Определить тип делегата.
  public delegate void CarEngineHandler(string msgForCaller);

  // 2. Определить переменную-член этого типа делегата.
  private CarEngineHandler _listOfHandlers;

  // 3. Добавить регистрационную функцию для вызывающего кода.
  public void RegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    _listOfHandlers = methodToCall;
  }
}
```

В приведенном примере обратите внимание на то, что типы делегатов определяются прямо внутри области действия класса `Car`; безусловно, это необязательно, но помогает закрепить идею о том, что делегат естественным образом работает с таким отдельным классом. Тип делегата `CarEngineHandler` может указывать на любой метод, который принимает значение `string` как параметр и имеет `void` в качестве возвращаемого типа.

Кроме того, была объявлена закрытая переменная-член делегата (`_listOfHandlers`) и вспомогательная функция (`RegisterWithCarEngine()`), которая позволяет вызывающему коду добавлять метод в список вызовов делегата.

На заметку! Строго говоря, переменную-член типа делегата можно было бы определить как `public`, избежав тем самым необходимости в создании дополнительных методов регистрации. Тем не менее, за счет определения этой переменной-члена типа делегата как `private` усиливается инкапсуляция и обеспечивается решение, более безопасное в отношении типов. Позже в главе при рассмотрении ключевого слова `event` языка C# мы еще вернемся к анализу рисков объявления переменных-членов с типами делегатов как `public`.

Теперь необходимо создать метод `Accelerate()`. Вспомните, что цель в том, чтобы позволить объекту `Car` отправлять связанные с двигателем сообщения любому подписавшемуся прослушивателю. Вот необходимое обновление:

```
// 4. Реализовать метод Accelerate() для обращения к списку
//    вызовов делегата в подходящих обстоятельствах.
public void Accelerate(int delta)
{
  // Если этот автомобиль сломан, то отправить сообщение об этом.
  if (_carIsDead)
  {
    _listOfHandlers?.Invoke("Sorry, this car is dead...");
  }
  else
  {
    CurrentSpeed += delta;

    // Автомобиль почти сломан?
```

```
    if (10 == (MaxSpeed - CurrentSpeed))
    {
      _listOfHandlers?.Invoke("Careful buddy! Gonna blow!");
    }
    if (CurrentSpeed >= MaxSpeed)
    {
      _carIsDead = true;
    }
    else
    {
      Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
    }
  }
}
```

Обратите внимание, что при попытке вызова методов, поддерживаемых переменной-членом `_listOfHandlers`, используется синтаксис распространения `null`. Причина в том, что создание таких объектов посредством вызова вспомогательного метода `RegisterWithCarEngine()` является задачей вызывающего кода. Если вызывающий код не вызывал `RegisterWithCarEngine()`, а мы попытаемся обратиться к списку вызовов делегата, то получим исключение `NullReferenceException` во время выполнения. Теперь, когда инфраструктура делегатов готова, внесите в файл `Program.cs` следующие изменения:

```
using System;
using CarDelegate;

Console.WriteLine("** Delegates as event enablers **\n");

// Создать объект Car.
Car c1 = new Car("SlugBug", 100, 10);

// Сообщить объекту Car, какой метод вызывать,
// когда он пожелает отправить сообщение.
c1.RegisterWithCarEngine(
  new Car.CarEngineHandler(OnCarEngineEvent));

// Увеличить скорость (это инициирует события).
Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}
Console.ReadLine();

// Цель для входящих сообщений.
static void OnCarEngineEvent(string msg)
{
  Console.WriteLine("\n*** Message From Car Object ***");
  Console.WriteLine("=> {0}", msg);
  Console.WriteLine("********************\n");
}
```

Код начинается с создания нового объекта `Car`. Поскольку вас интересуют события, связанные с двигателем, следующий шаг заключается в вызове специальной регистрационной функции `RegisterWithCarEngine()`. Вспомните, что метод `RegisterWithCarEngine()` ожидает получения экземпляра вложенного делегата

CarEngineHandler, и как в случае любого делегата, в параметре конструктора передается метод, на который он должен указывать. Трюк здесь в том, что интересующий метод находится в классе Program! Обратите также внимание, что метод OnCarEngineEvent() полностью соответствует связанному делегату, потому что принимает string и возвращает void. Ниже показан вывод приведенного примера:

```
***** Delegates as event enablers *****
***** Speeding up *****
CurrentSpeed = 30
CurrentSpeed = 50
CurrentSpeed = 70
***** Message From Car Object *****
=> Careful buddy! Gonna blow!
**********************************
CurrentSpeed = 90
***** Message From Car Object *****
=> Sorry, this car is dead...
**********************************
```

Включение группового вызова

Вспомните, что делегаты .NET Core обладают встроенной возможностью *группового вызова*. Другими словами, объект делегата может поддерживать целый список методов для вызова, а не просто единственный метод. Для добавления нескольких методов к объекту делегата вместо прямого присваивания применяется перегруженная операция +=. Чтобы включить групповой вызов в классе Car, можно модифицировать метод RegisterWithCarEngine():

```
public class Car
{
  // Добавление поддержки группового вызова.
  // Обратите внимание на использование операции +=,
  // а не обычной операции присваивания (=).
  public void RegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    _listOfHandlers += methodToCall;
  }
  ...
}
```

Когда операция += используется с объектом делегата, компилятор преобразует ее в вызов статического метода Delegate.Combine(). На самом деле можно было бы вызывать Delegate.Combine() напрямую, однако операция += предлагает более простую альтернативу. Хотя нет никакой необходимости в модификации текущего метода RegisterWithCarEngine(), ниже представлен пример применения Delegate.Combine() вместо операции +=:

```
public void RegisterWithCarEngine( CarEngineHandler methodToCall )
{
  if (_listOfHandlers == null)
  {
    _listOfHandlers = methodToCall;
  }
  else
```

```
  {
    _listOfHandlers =
      Delegate.Combine(_listOfHandlers, methodToCall)
        as CarEngineHandler;
  }
}
```

В любом случае вызывающий код теперь может регистрировать множественные цели для одного и того же обратного вызова. Второй обработчик выводит входное сообщение в верхнем регистре просто ради отображения:

```
Console.WriteLine("***** Delegates as event enablers *****\n");

// Создать объект Car.
Car c1 = new Car("SlugBug", 100, 10);

// Зарегистрировать несколько обработчиков событий.
c1.RegisterWithCarEngine(
  new Car.CarEngineHandler(OnCarEngineEvent));
c1.RegisterWithCarEngine(
  new Car.CarEngineHandler(OnCarEngineEvent2));

// Увеличить скорость (это инициирует события).
Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}
Console.ReadLine();

// Теперь есть ДВА метода, которые будут
// вызываться Car при отправке уведомлений.
static void OnCarEngineEvent(string msg)
{
  Console.WriteLine("\n*** Message From Car Object ***");
  Console.WriteLine("=> {0}", msg);
  Console.WriteLine("********************************\n");
}

static void OnCarEngineEvent2(string msg)
{
  Console.WriteLine("=> {0}", msg.ToUpper());
}
```

Удаление целей из списка вызовов делегата

В классе `Delegate` также определен статический метод `Remove()`, который позволяет вызывающему коду динамически удалять отдельные методы из списка вызовов объекта делегата. В итоге у вызывающего кода появляется возможность легко "отменять подписку" на заданное уведомление во время выполнения. Хотя метод `Delegate.Remove()` допускается вызывать в коде напрямую, разработчики C# могут использовать в качестве удобного сокращения операцию `-=`. Давайте добавим в класс `Car` новый метод, который позволяет вызывающему коду исключать метод из списка вызовов:

```
public class Car
{
  ...
```

```
  public void UnRegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    _listOfHandlers -= methodToCall;
  }
}
```

При таком обновлении класса `Car` прекратить получение уведомлений от второго обработчика можно за счет изменения вызывающего кода следующим образом:

```
Console.WriteLine("***** Delegates as event enablers *****\n");

// Создать объект Car.
Car c1 = new Car("SlugBug", 100, 10);
c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

// На этот раз сохранить объект делегата, чтобы позже
// можно было отменить регистрацию.
Car.CarEngineHandler handler2 =
  new Car.CarEngineHandler(OnCarEngineEvent2);
c1.RegisterWithCarEngine(handler2);

// Увеличить скорость (это инициирует события).
Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}

// Отменить регистрацию второго обработчика.
c1.UnRegisterWithCarEngine(handler2);

// Сообщения в верхнем регистре больше не выводятся.
Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}

Console.ReadLine();
```

Отличие этого кода в том, что здесь создается объект `Car.CarEngineHandler`, который сохраняется в локальной переменной, чтобы впоследствии можно было отменить подписку на получение уведомлений. Таким образом, при увеличении скорости объекта `Car` во второй раз версия входного сообщения в верхнем регистре больше выводиться не будет, поскольку данная цель исключена из списка вызовов делегата.

Синтаксис групповых преобразований методов

В предыдущем примере `CarDelegate` явно создавались экземпляры класса делегата `Car.CarEngineHandler` для регистрации и отмены регистрации на получение уведомлений:

```
Console.WriteLine("***** Delegates as event enablers *****\n");

Car c1 = new Car("SlugBug", 100, 10);
c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

Car.CarEngineHandler handler2 =
  new Car.CarEngineHandler(OnCarEngineEvent2);
c1.RegisterWithCarEngine(handler2);
...
```

Конечно, если необходимо вызывать любые унаследованные члены класса `MulticastDelegate` или `Delegate`, то проще всего вручную создать переменную делегата. Однако в большинстве случаев иметь дело с внутренним устройством объекта делегата не требуется. Объект делегата обычно придется применять только для передачи имени метода в параметре конструктора.

Для простоты в языке C# предлагается сокращение, называемое *групповым преобразованием методов*. Это средство позволяет указывать вместо объекта делегата прямое имя метода, когда вызываются методы, которые принимают делегаты в качестве аргументов.

На заметку! Позже в главе вы увидите, что синтаксис группового преобразования методов можно также использовать для упрощения регистрации событий C#.

В целях иллюстрации внесите в файл `Program.cs` показанные ниже изменения, где групповое преобразование методов применяется для регистрации и отмены регистрации подписки на уведомления:

```
Console.WriteLine("***** Method Group Conversion *****\n");
Car c2 = new Car();

// Зарегистрировать простое имя метода.
c2.RegisterWithCarEngine(OnCarEngineEvent);
Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
{
  c2.Accelerate(20);
}

// Отменить регистрацию простого имени метода.
c2.UnRegisterWithCarEngine(OnCarEngineEvent);

// Уведомления больше не поступают!
for (int i = 0; i < 6; i++)
{
  c2.Accelerate(20);
}

Console.ReadLine();
```

Обратите внимание, что мы не создаем напрямую ассоциированный объект делегата, а просто указываем метод, который соответствует ожидаемой сигнатуре делегата (в данном случае метод, возвращающий `void` и принимающий единственный аргумент `string`). Имейте в виду, что компилятор C# по-прежнему обеспечивает безопасность в отношении типов. Таким образом, если метод `OnCarEngineEvent()` не принимает `string` и не возвращает `void`, тогда возникнет ошибка на этапе компиляции.

Понятие обобщенных делегатов

В главе 10 упоминалось о том, что язык C# позволяет определять обобщенные типы делегатов. Например, предположим, что необходимо определить тип делегата, который может вызывать любой метод, возвращающий `void` и принимающий единственный параметр. Если передаваемый аргумент может изменяться, то это легко смоделировать с использованием параметра типа. Взгляните на следующий код внутри нового проекта консольного приложения по имени `GenericDelegate`:

```
Console.WriteLine("***** Generic Delegates *****\n");

// Зарегистрировать цели.
MyGenericDelegate<string> strTarget =
  new MyGenericDelegate<string>(StringTarget);
strTarget("Some string data");

// Использовать синтаксис группового преобразования методов.
MyGenericDelegate<int> intTarget = IntTarget;
intTarget(9);
Console.ReadLine();

static void StringTarget(string arg)
{
  Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}

static void IntTarget(int arg)
{
  Console.WriteLine("++arg is: {0}", ++arg);
}

// Этот обобщенный делегат может вызывать любой метод, который
// возвращает void и принимает единственный параметр типа T.
public delegate void MyGenericDelegate<T>(T arg);
```

Как видите, в типе делегата `MyGenericDelegate<T>` определен единственный параметр, представляющий аргумент для передачи цели делегата. При создании экземпляра этого типа должно быть указано значение параметра типа наряду с именем метода, который делегат может вызывать. Таким образом, если указать тип `string`, тогда целевому методу будет отправляться строковое значение:

```
// Создать экземпляр MyGenericDelegate<T>
// с указанием string в качестве параметра типа.
MyGenericDelegate<string> strTarget = StringTarget;
strTarget("Some string data");
```

С учетом формата объекта `strTarget` метод `StringTarget` теперь должен принимать в качестве параметра единственную строку:

```
static void StringTarget(string arg)
{
  Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}
```

Обобщенные делегаты `Action<>` и `Func<>`

В настоящей главе вы уже видели, что когда нужно применять делегаты для обратных вызовов в приложениях, обычно должны быть выполнены описанные далее шаги.

1. Определить специальный делегат, соответствующий формату метода, на который он указывает.
2. Создать экземпляр специального делегата, передав имя метода в качестве аргумента конструктора.
3. Косвенно обратиться к методу через вызов `Invoke()` на объекте делегата.

В случае принятия такого подхода в итоге, как правило, получается несколько специальных делегатов, которые могут никогда не использоваться за рамками текущей задачи (например, MyGenericDelegate<T>, CarEngineHandler и т.д.). Хотя вполне может быть и так, что для проекта требуется специальный уникально именованный делегат, в других ситуациях точное *имя* типа делегата роли не играет. Во многих случаях просто необходим "какой-нибудь делегат", который принимает набор аргументов и возможно возвращает значение, отличное от void. В таких ситуациях можно применять встроенные в инфраструктуру делегаты Action<> и Func<>. Чтобы удостовериться в их полезности, создайте новый проект консольного приложения по имени ActionAndFuncDelegates.

Обобщенный делегат Action<> определен в пространствах имен System. Его можно использовать для "указания" на метод, который принимает вплоть до *16 аргументов* (чего должно быть вполне достаточно!) и возвращает void. Вспомните, что поскольку Action<> является обобщенным делегатом, понадобится также указывать типы всех параметров.

Модифицируйте класс Program, определив в нем новый статический метод, который принимает (скажем) три уникальных параметра:

```
// Это цель для делегата Action<>.
static void DisplayMessage(string msg, ConsoleColor txtColor,
                           int printCount)
{
  // Установить цвет текста консоли.
  ConsoleColor previous = Console.ForegroundColor;
  Console.ForegroundColor = txtColor;

  for (int i = 0; i < printCount; i++)
  {
    Console.WriteLine(msg);
  }

  // Восстановить цвет.
  Console.ForegroundColor = previous;
}
```

Теперь вместо построения специального делегата вручную для передачи потока программы методу DisplayMessage() вы можете применять готовый делегат Action<>:

```
Console.WriteLine("***** Fun with Action and Func *****");

// Использовать делегат Action<> для указания на метод DisplayMessage().
Action<string, ConsoleColor, int> actionTarget = DisplayMessage;
actionTarget("Action Message!", ConsoleColor.Yellow, 5);

Console.ReadLine();
```

Как видите, при использовании делегата Action<> не нужно беспокоиться об определении специального типа делегата. Тем не менее, как уже упоминалось, тип делегата Action<> позволяет указывать только на методы, возвращающие void. Если необходимо указывать на метод, имеющий возвращаемое значение (и нет желания заниматься написанием собственного типа делегата), тогда можно применять тип делегата Func<>.

Обобщенный делегат Func<> способен указывать на методы, которые (подобно Action<>) принимают вплоть до 16 параметров и имеют специальное возвращаемое значение. В целях иллюстрации добавьте в класс Program новый метод:

```
// Цель для делегата Func<>.
static int Add(int x, int y)
{
  return x + y;
}
```

Ранее в главе был построен специальный делегат BinaryOp для "указания" на методы сложения и вычитания. Теперь задачу можно упростить за счет использования версии Func<>, которая принимает всего три параметра типа. Учтите, что *последний* параметр в Func<> *всегда* представляет возвращаемое значение метода. Чтобы закрепить данный момент, предположим, что в классе Program также определен следующий метод:

```
static string SumToString(int x, int y)
{
  return (x + y).ToString();
}
```

Вызовите эти методы:

```
Func<int, int, int> funcTarget = Add;
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = SumToString;
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

С учетом того, что делегаты Action<> и Func<> могут устранить шаг по ручному определению специального делегата, вас может интересовать, должны ли они применяться всегда. Подобно большинству аспектов программирования ответ таков: в зависимости от ситуации. Во многих случаях Action<> и Func<> будут предпочтительным вариантом. Однако если необходим делегат со специальным именем, которое, как вам кажется, помогает лучше отразить предметную область, то построение специального делегата сводится к единственному оператору кода. В оставшихся материалах книги вы увидите оба подхода.

На заметку! Делегаты Action<> и Func<> интенсивно используются во многих важных API-интерфейсах .NET Core, включая инфраструктуру параллельного программирования и LINQ (помимо прочих).

Итак, первоначальный экскурс в типы делегатов окончен. Теперь давайте перейдем к обсуждению связанной темы — ключевого слова event языка C#.

Понятие событий C#

Делегаты — довольно интересные конструкции в том плане, что позволяют объектам, находящимся в памяти, участвовать в двустороннем взаимодействии. Тем не менее, прямая работа с делегатами может приводить к написанию стереотипного кода (определение делегата, определение необходимых переменных-членов, создание

специальных методов регистрации и отмены регистрации для предохранения инкапсуляции и т.д.).

Более того, во время применения делегатов непосредственным образом как механизма обратного вызова в приложениях, если вы не определите переменную-член типа делегата в классе как закрытую, тогда вызывающий код будет иметь прямой доступ к объектам делегатов. В таком случае вызывающий код может присвоить переменной-члену новый объект делегата (фактически удаляя текущий список функций, которые подлежат вызову) и, что даже хуже, вызывающий код сможет напрямую обращаться к списку вызовов делегата. В целях демонстрации создайте новый проект консольного приложения по имени `PublicDelegateProblem` и добавьте следующую переделанную (и упрощенную) версию класса `Car` из предыдущего примера `CarDelegate`:

```
namespace PublicDelegateproblem
{
  public class Car
  {
    public delegate void CarEngineHandler(string msgForCaller);
    // Теперь это член public!
    public CarEngineHandler ListOfHandlers;
    // Просто вызвать уведомление Exploded.
    public void Accelerate(int delta)
    {
      if (ListOfHandlers != null)
      {
        ListOfHandlers("Sorry, this car is dead...");
      }
    }
  }
}
```

Обратите внимание, что у вас больше нет закрытых переменных-членов с типами делегатов, инкапсулированных с помощью специальных методов регистрации. Поскольку эти члены на самом деле открытые, вызывающий код может получить доступ прямо к переменной-члену `listOfHandlers`, присвоить ей новые объекты `CarEngineHandler` и вызвать делегат по своему желанию:

```
using System;
using PublicDelegateProblem;
Console.WriteLine("***** Agh! No Encapsulation! *****\n");
// Создать объект Car.
Car myCar = new Car();
// Есть прямой доступ к делегату!
myCar.ListOfHandlers = CallWhenExploded;
myCar.Accelerate(10);
// Теперь можно присвоить полностью новый объект...
// что в лучшем случае сбивает с толку.
myCar.ListOfHandlers = CallHereToo;
myCar.Accelerate(10);
// Вызывающий код может также напрямую вызывать делегат!
myCar.ListOfHandlers.Invoke("hee, hee, hee...");
Console.ReadLine();
```

```
static void CallWhenExploded(string msg)
{
  Console.WriteLine(msg);
}
static void CallHereToo(string msg)
{
  Console.WriteLine(msg);
}
```

Открытие доступа к членам типа делегата нарушает инкапсуляцию, что не только затруднит сопровождение кода (и отладку), но также сделает приложение уязвимым в плане безопасности! Ниже показан вывод текущего примера:

```
***** Agh! No Encapsulation! *****
Sorry, this car is dead...
Sorry, this car is dead...
hee, hee, hee...
```

Очевидно, что вы не захотите предоставлять другим приложениям возможность изменять то, на что указывает делегат, или вызывать его члены без вашего разрешения. С учетом сказанного общепринятая практика предусматривает объявление переменных-членов, имеющих типы делегатов, как закрытых.

Ключевое слово `event`

В качестве сокращения, избавляющего от необходимости создавать специальные методы для добавления и удаления методов из списка вызовов делегата, в языке C# предлагается ключевое слово `event`. В результате обработки компилятором ключевого слова `event` вы автоматически получаете методы регистрации и отмены регистрации, а также все необходимые переменные-члены для типов делегатов. Такие переменные-члены с типами делегатов *всегда* объявляются как закрытые и потому они не доступны напрямую из объекта, инициирующего событие. В итоге ключевое слово `event` может использоваться для упрощения отправки специальным классом уведомлений внешним объектам.

Определение события представляет собой двухэтапный процесс. Во-первых, понадобится определить тип делегата (или задействовать существующий тип), который будет хранить список методов, подлежащих вызову при возникновении события. Во-вторых, необходимо объявить событие (с применением ключевого слова `event`) в терминах связанного типа делегата.

Чтобы продемонстрировать использование ключевого слова `event`, создайте новый проект консольного приложения по имени `CarEvents`. В этой версии класса `Car` будут определены два события под названиями `AboutToBlow` и `Exploded`, которые ассоциированы с единственным типом делегата по имени `CarEngineHandler`. Ниже показаны начальные изменения, внесенные в класс `Car`:

```
using System;
namespace CarEvents
{
  public class Car
  {
    ...
    // Этот делегат работает в сочетании с событиями Car.
    public delegate void CarEngineHandler(string msgForCaller);
```

```
    // Этот объект Car может отправлять следующие события:
    public event CarEngineHandler Exploded;
    public event CarEngineHandler AboutToBlow;
    ...
  }
}
```

Отправка события вызывающему коду сводится просто к указанию события по имени наряду со всеми обязательными параметрами, как определено ассоциированным делегатом. Чтобы удостовериться в том, что вызывающий код действительно зарегистрировал событие, перед вызовом набора методов делегата событие следует проверить на равенство `null`. Ниже приведена новая версия метода `Accelerate()` класса `Car`:

```
public void Accelerate(int delta)
{
  // Если автомобиль сломан, то инициировать событие Exploded.
  if (_carIsDead)
  {
    Exploded?.Invoke("Sorry, this car is dead...");
  }
  else
  {
    CurrentSpeed += delta;
    // Почти сломан?
    if (10 == MaxSpeed - CurrentSpeed)
    {
      AboutToBlow?.Invoke("Careful buddy! Gonna blow!");
    }
    // Все еще в порядке!
    if (CurrentSpeed >= MaxSpeed)
    {
      _carIsDead = true;
    }
    else
    {
      Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
    }
  }
}
```

Итак, класс `Car` был сконфигурирован для отправки двух специальных событий без необходимости в определении специальных функций регистрации или в объявлении переменных-членов, имеющих типы делегатов. Применение нового объекта вы увидите очень скоро, но сначала давайте чуть подробнее рассмотрим архитектуру событий.

"За кулисами" событий

Когда компилятор C# обрабатывает ключевое слово `event`, он генерирует два скрытых метода, один с префиксом `add_`, а другой с префиксом `remove_`. За префиксом следует имя события C#. Например, событие `Exploded` дает в результате два скрытых метода с именами `add_Exploded()` и `remove_Exploded()`. Если заглянуть в

код CIL метода add_AboutToBlow(), то можно обнаружить вызов метода Delegate.Combine(). Взгляните на частичный код CIL:

```
.method public hidebysig specialname instance void  add_AboutToBlow(
  class [System.Runtime]System.EventHandler`1<class CarEvents.
CarEventArgs> 'value') cil
  managed
  {
    ...
    IL_000b: call class [System.Runtime]System.Delegate [System.
Runtime]System.
    Delegate::Combine(class [System.Runtime]System.Delegate,
                      class [System.Runtime]System.
    Delegate)
    ...
  } // end of method Car::add_AboutToBlow
```

Как и можно было ожидать, метод remove_AboutToBlow() будет вызывать Delegate.Remove():

```
.method public hidebysig specialname instance void  remove_AboutToBlow (
  class [System.Runtime]System.EventHandler`1
<class CarEvents.CarEventArgs> 'value') cil
  managed
  {
    ...
    IL_000b: call class [System.Runtime]System.Delegate [System.
Runtime]System.
    Delegate::Remove(class [System.Runtime]System.Delegate,
                     class [System.Runtime]System.
    Delegate)
  ...
}
```

Наконец, в коде CIL, представляющем само событие, используются директивы .addon и .removeon для отображения на имена корректных методов add_XXX() и remove_XXX(), подлежащих вызову:

```
.event class [System.Runtime]System.EventHandler`1<class CarEvents.
CarEventArgs> AboutToBlow
{
  .addon instance void CarEvents.Car::add_AboutToBlow(
    class [System.Runtime]System.EventHandler`1<class CarEvents.
CarEventArgs>)
  .removeon instance void CarEvents.Car::remove_AboutToBlow(
    class [System.Runtime]System.EventHandler`1
<class CarEvents.CarEventArgs>)
} // end of event Car::AboutToBlow
```

Теперь, когда вы понимаете, каким образом строить класс, способный отправлять события C# (и знаете, что события — всего лишь способ сэкономить время на наборе кода), следующий крупный вопрос связан с организацией прослушивания входящих событий на стороне вызывающего кода.

Прослушивание входящих событий

События C# также упрощают действие по регистрации обработчиков событий на стороне вызывающего кода. Вместо того чтобы указывать специальные вспомогательные методы, вызывающий код просто применяет операции += и -= напрямую (что приводит к внутренним вызовам методов add_XXX() или remove_XXX()). При регистрации события руководствуйтесь показанным ниже шаблоном:

```
// ИмяОбъекта.ИмяСобытия +=
//   new СвязанныйДелегат(функцияДляВызова);
Car.CarEngineHandler d =
  new Car.CarEngineHandler(CarExplodedEventHandler);
myCar.Exploded += d;
```

Отключить от источника событий можно с помощью операции -= в соответствии со следующим шаблоном:

```
// ИмяОбъекта.ИмяСобытия -=
//   СвязанныйДелегат(функцияДляВызова);
myCar.Exploded -= d;
```

Кроме того, с событиями можно использовать синтаксис группового преобразования методов:

```
Car.CarEngineHandler d = CarExplodedEventHandler;
myCar.Exploded += d;
```

При наличии таких весьма предсказуемых шаблонов переделайте вызывающий код, применив на этот раз синтаксис регистрации событий C#:

```
Console.WriteLine("***** Fun with Events *****\n");
Car c1 = new Car("SlugBug", 100, 10);

// Зарегистрировать обработчики событий.
c1.AboutToBlow += CarIsAlmostDoomed;
c1.AboutToBlow += CarAboutToBlow;

Car.CarEngineHandler d = CarExploded;
c1.Exploded += d;

Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}

// Удалить метод CarExploded() из списка вызовов.
c1.Exploded -= d;

Console.WriteLine("\n***** Speeding up *****");
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}
Console.ReadLine();

static void CarAboutToBlow(string msg)
{
  Console.WriteLine(msg);
}
```

```
static void CarIsAlmostDoomed(string msg)
{
  Console.WriteLine("=> Critical Message from Car: {0}", msg);
}
static void CarExploded(string msg)
{
  Console.WriteLine(msg);
}
```

Упрощение регистрации событий с использованием Visual Studio

Среда Visual Studio предлагает помощь в процессе регистрации обработчиков событий. В случае применения синтаксиса += при регистрации событий открывается окно IntelliSense, приглашающее нажать клавишу <Tab> для автоматического завершения связанного экземпляра делегата (рис. 12.1), что достигается с использованием синтаксиса групповых преобразований методов.

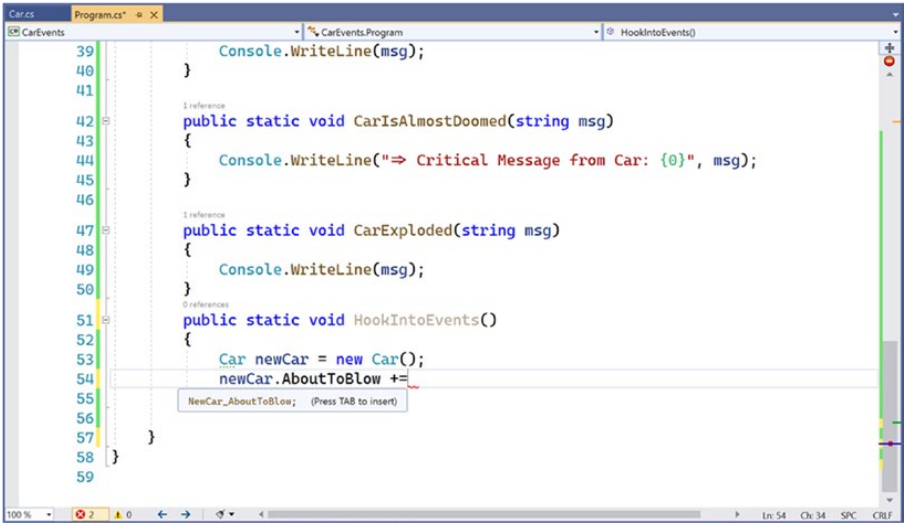

Рис. 12.1. Выбор делегата с помощью средства IntelliSense

После нажатия клавиши <Tab> будет сгенерирован новый метод, как показано на рис. 12.2.

Обратите внимание, что код заглушки имеет корректный формат цели делегата (кроме того, метод объявлен как `static`, т.к. событие было зарегистрировано внутри статического метода):

```
static void NewCar_AboutToBlow(string msg)
{
  throw new NotImplementedException();
}
```

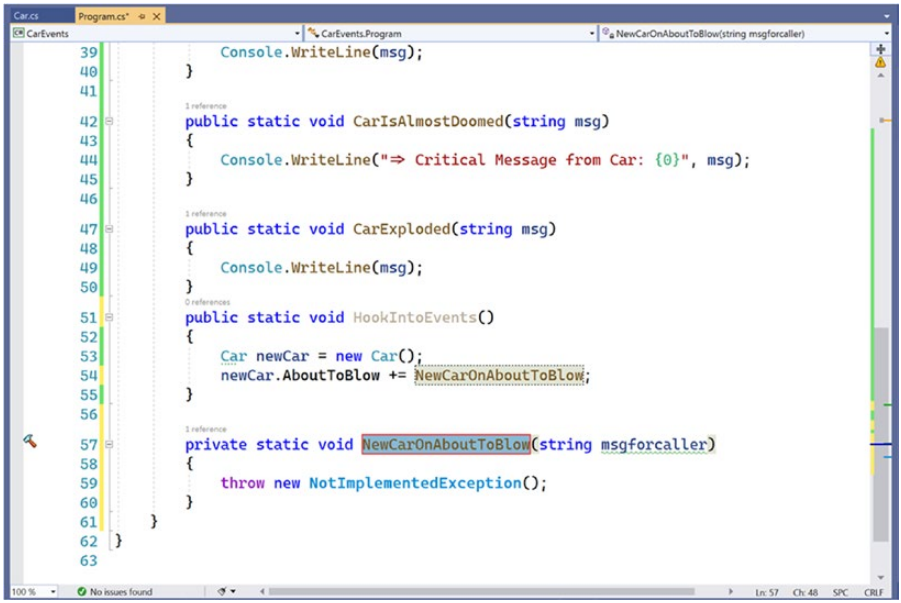

Рис. 12.2. Оформление цели делегата средством IntelliSense

Средство IntelliSense доступно для всех событий .NET Core, ваших событий и событий из библиотек базовых классов. Такая возможность IDE-среды значительно экономит время, избавляя от необходимости выяснять с помощью справочной системы подходящий тип делегата для применения с заданным событием и формат целевого метода делегата.

Создание специальных аргументов событий

По правде говоря, в текущую итерацию класса `Car` можно было бы внести последнее усовершенствование, которое отражает рекомендованный Microsoft шаблон событий. Если вы начнете исследовать события, отправляемые определенным типом из библиотек базовых классов, то обнаружите, что первый параметр лежащего в основе делегата имеет тип `System.Object`, в то время как второй — тип, производный от `System.EventArgs`.

Параметр `System.Object` представляет ссылку на объект, который отправляет событие (такой как `Car`), а второй параметр — информацию, относящуюся к обрабатываемому событию. Базовый класс `System.EventArgs` представляет событие, которое не сопровождается какой-либо специальной информацией:

```
public class EventArgs
{
  public static readonly EventArgs Empty;
  public EventArgs();
}
```

Для простых событий экземпляр EventArgs можно передать напрямую. Тем не менее, когда нужно передавать специальные данные, вы должны построить подходящий класс, производный от EventArgs. В этом примере предположим, что есть класс по имени CarEventArgs, который поддерживает строковое представление сообщения, отправленного получателю:

```
using System;

namespace CarEvents
{
  public class CarEventArgs : EventArgs
  {
    public readonly string msg;
    public CarEventArgs(string message)
    {
      msg = message;
    }
  }
}
```

Теперь можно модифицировать тип делегата CarEngineHandler, как показано ниже (события не изменятся):

```
public class Car
{
  public delegate void CarEngineHandler(object sender, CarEventArgs e);
  ...
}
```

Здесь при инициировании событий внутри метода Accelerate() необходимо использовать ссылку на текущий объект Car (посредством ключевого слова this) и экземпляр типа CarEventArgs. Например, рассмотрим следующее обновление:

```
public void Accelerate(int delta)
{
  // Если этот автомобиль сломан, то инициировать событие Exploded.
  if (carIsDead)
  {
    Exploded?.Invoke(this, new CarEventArgs("Sorry, this car is dead..."));
  }
  ...
}
```

На вызывающей стороне понадобится лишь модифицировать обработчики событий для приема входных параметров и получения сообщения через поле, доступное только для чтения. Вот пример:

```
static void CarAboutToBlow(object sender, CarEventArgs e)
{
  Console.WriteLine($"{sender} says: {e.msg}");
}
```

Если получатель желает взаимодействовать с объектом, отправившим событие, тогда можно выполнить явное приведение System.Object. Такая ссылка позволит вызывать любой открытый метод объекта, который отправил уведомление:

```
static void CarAboutToBlow(object sender, CarEventArgs e)
{
  // Просто для подстраховки перед приведением
  // произвести проверку во время выполнения.
  if (sender is Car c)
  {
    Console.WriteLine(
      $"Critical Message from {c.PetName}: {e.msg}");
  }
}
```

Обобщенный делегат EventHandler<T>

С учетом того, что очень многие специальные делегаты принимают экземпляр `object` в первом параметре и экземпляр производного от `EventArgs` класса во втором, предыдущий пример можно дополнительно упростить за счет применения обобщенного типа `EventHandler<T>`, где `T` — специальный тип, производный от `EventArgs`. Рассмотрим следующую модификацию типа `Car` (обратите внимание, что определять специальный тип делегата больше не нужно):

```
public class Car
{
  ...
  public event EventHandler<CarEventArgs> Exploded;
  public event EventHandler<CarEventArgs> AboutToBlow;
}
```

Затем в вызывающем коде тип `EventHandler<CarEventArgs>` можно использовать везде, где ранее указывался `CarEngineHandler` (или снова применять групповое преобразование методов):

```
Console.WriteLine("***** Prim and Proper Events *****\n");

// Создать объект Car обычным образом.
Car c1 = new Car("SlugBug", 100, 10);

// Зарегистрировать обработчики событий.
c1.AboutToBlow += CarIsAlmostDoomed;
c1.AboutToBlow += CarAboutToBlow;

EventHandler<CarEventArgs> d = CarExploded;
c1.Exploded += d;
...
```

Итак, к настоящему моменту вы узнали основные аспекты работы с делегатами и событиями в C#. Хотя этого вполне достаточно для решения практически любых задач, связанных с обратными вызовами, в завершение главы мы рассмотрим несколько финальных упрощений, в частности анонимные методы и лямбда-выражения.

Понятие анонимных методов C#

Как было показано ранее, когда вызывающий код желает прослушивать входящие события, он должен определить специальный метод в классе (или структуре), который соответствует сигнатуре ассоциированного делегата. Ниже приведен пример:

```
SomeType t = new SomeType();

// Предположим, что SomeDeletage может указывать на методы,
// которые не принимают аргументов и возвращают void.
t.SomeEvent += new SomeDelegate(MyEventHandler);

// Обычно вызывается только объектом SomeDelegate.
static void MyEventHandler()
{
  // Делать что-нибудь при возникновении события.
}
```

Однако если подумать, то такие методы, как `MyEventHandler()`, редко предназначены для вызова из любой другой части программы кроме делегата. С точки зрения продуктивности вручную определять отдельный метод для вызова объектом делегата несколько хлопотно (хотя и вполне допустимо).

Для решения указанной проблемы событие можно ассоциировать прямо с блоком операторов кода во время регистрации события. Формально такой код называется *анонимным методом*. Чтобы ознакомиться с синтаксисом, создайте новый проект консольного приложения по имени `AnonymousMethods`, после чего скопируйте в него файлы `Car.cs` и `CarEventArgs.cs` из проекта `CarEvents` (не забыв изменить пространство имен на `AnonymousMethods`). Модифицируйте код в файле `Program.cs`, как показано ниже, для обработки событий, посылаемых из класса `Car`, с использованием анонимных методов вместо специальных именованных обработчиков событий:

```
using System;
using AnonymousMethods;

Console.WriteLine("***** Anonymous Methods *****\n");
Car c1 = new Car("SlugBug", 100, 10);

// Зарегистрировать обработчики событий как анонимные методы.
c1.AboutToBlow += delegate
{
  Console.WriteLine("Eek! Going too fast!");
};

c1.AboutToBlow += delegate(object sender, CarEventArgs e)
{
  Console.WriteLine("Message from Car: {0}", e.msg);
};

c1.Exploded += delegate(object sender, CarEventArgs e)
{
  Console.WriteLine("Fatal Message from Car: {0}", e.msg);
};

// В конце концов, этот код будет инициировать события.
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}
Console.ReadLine();
```

На заметку! После финальной фигурной скобки в анонимном методе должна быть помещена точка с запятой, иначе возникнет ошибка на этапе компиляции.

И снова легко заметить, что специальные статические обработчики событий вроде `CarAboutToBlow()` или `CarExploded()` в вызывающем коде больше не определяются. Взамен с помощью синтаксиса += определяются встроенные неименованные (т.е. анонимные) методы, к которым вызывающий код будет обращаться во время обработки события. Базовый синтаксис анонимного метода представлен следующим псевдокодом:

```
НекоторыйТип t = new НекоторыйТип();
t.НекотороеСобытие += delegate (необязательноУказываемыеАргументыДелегата)
{ /* операторы */ };
```

Обратите внимание, что при обработке первого события `AboutToBlow` внутри предыдущего примера кода аргументы, передаваемые из делегата, не указывались:

```
c1.AboutToBlow += delegate
{
  Console.WriteLine("Eek! Going too fast!");
};
```

Строго говоря, вы не обязаны принимать входные аргументы, отправленные специфическим событием. Но если вы хотите задействовать эти входные аргументы, тогда понадобится указать параметры, прототипированные типом делегата (как показано во второй обработке событий `AboutToBlow` и `Exploded`). Например:

```
c1.AboutToBlow += delegate(object sender, CarEventArgs e)
{
  Console.WriteLine("Critical Message from Car: {0}", e.msg);
};
```

Доступ к локальным переменным

Анонимные методы интересны тем, что способны обращаться к локальным переменным метода, где они определены. Формально такие переменные называются *внешними переменными* анонимного метода. Ниже перечислены важные моменты, касающиеся взаимодействия между областью действия анонимного метода и областью действия метода, в котором он определен.

- Анонимный метод не имеет доступа к параметрам `ref` и `out` определяющего метода.
- Анонимный метод не может иметь локальную переменную, имя которой совпадает с именем локальной переменной внешнего метода.
- Анонимный метод может обращаться к переменным экземпляра (или статическим переменным) из области действия внешнего класса.
- Анонимный метод может объявлять локальную переменную с тем же именем, что и у переменной-члена внешнего класса (локальные переменные имеют отдельную область действия и скрывают переменные-члены из внешнего класса).

Предположим, что в операторах верхнего уровня определена локальная переменная по имени `aboutToBlowCounter` типа `int`. Внутри анонимных методов, которые обрабатывают событие `AboutToBlow`, выполните увеличение значения `aboutToBlowCounter` на 1 и вывод результата на консоль перед завершением операторов:

```
Console.WriteLine("***** Anonymous Methods *****\n");
int aboutToBlowCounter = 0;

// Создать объект Car обычным образом.
Car c1 = new Car("SlugBug", 100, 10);

// Зарегистрировать обработчики событий как анонимные методы.
c1.AboutToBlow += delegate
{
  aboutToBlowCounter++;
  Console.WriteLine("Eek! Going too fast!");
};

c1.AboutToBlow += delegate(object sender, CarEventArgs e)
{
  aboutToBlowCounter++;
  Console.WriteLine("Critical Message from Car: {0}", e.msg);
};
...
// В конце концов, это будет инициировать события.
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}

Console.WriteLine("AboutToBlow event was fired {0} times.",
  aboutToBlowCounter);
Console.ReadLine();
```

После запуска модифицированного кода вы обнаружите, что финальный вывод `Console.WriteLine()` сообщает о двукратном инициировании события `AboutToBlow`.

Использование ключевого слова `static` с анонимными методами (нововведение в версии 9.0)

В предыдущем примере демонстрировались анонимные методы, которые взаимодействовали с переменными, объявленными вне области действия самих методов. Хотя возможно именно это входило в ваши намерения, прием нарушает инкапсуляцию и может привести к нежелательным побочным эффектам в программе. Вспомните из главы 4, что локальные функции могут быть изолированы от содержащего их кода за счет их настройки как статических, например:

```
static int AddWrapperWithStatic(int x, int y)
{
  // Выполнить проверку достоверности.
  return Add(x,y);
  static int Add(int x, int y)
  {
    return x + y;
  }
}
```

В версии C# 9.0 анонимные методы также могут быть помечены как статические с целью предохранения инкапсуляции и гарантирования того, что они не привнесут

какие-либо побочные эффекты в код, где они содержатся. Вот как выглядит модифицированный анонимный метод:

```
c1.AboutToBlow += static delegate
{
  // Этот код приводит к ошибке на этапе компиляции,
  // потому что анонимный метод помечен как статический.
  aboutToBlowCounter++;
  Console.WriteLine("Eek! Going too fast!");
};
```

Предыдущий код не скомпилируется из-за попытки анонимного метода получить доступ к переменной, объявленной вне области его действия.

Использование отбрасывания с анонимными методами (нововведение в версии 9.0)

Отбрасывание, представленное в главе 3, в версии C# 9.0 было обновлено с целью применения в качестве входных параметров, но с одной уловкой. Поскольку символ подчеркивания (_) в предшествующих версиях C# считался законным идентификатором переменной, в анонимном методе должно присутствовать два и более подчеркиваний, чтобы они трактовались как отбрасывание.

Например, в следующем коде создается делегат `Func<>`, который принимает два целых числа и возвращает еще одно целое число. Приведенная реализация игнорирует любые переданные переменные и возвращает значение `42`:

```
Console.WriteLine("******** Discards with Anonymous Methods ********");

Func<int,int,int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine("constant(3,4)={0}",constant(3,4));
```

Понятие лямбда-выражений

Чтобы завершить знакомство с архитектурой событий .NET Core, необходимо исследовать *лямбда-выражения*. Как объяснялось ранее в главе, язык C# поддерживает возможность обработки событий "встраиваемым образом", позволяя назначать блок операторов кода прямо событию с применением анонимных методов вместо построения отдельного метода, который должен вызываться делегатом. Лямбда-выражения — всего лишь лаконичный способ записи анонимных методов, который в конечном итоге упрощает работу с типами делегатов .NET Core.

В целях подготовки фундамента для изучения лямбда-выражений создайте новый проект консольного приложения по имени `LambdaExpressions`. Для начала взгляните на метод `FindAll()` обобщенного класса `List<T>`. Данный метод можно вызывать, когда нужно извлечь подмножество элементов из коллекции; вот его прототип:

```
// Метод класса System.Collections.Generic.List<T>.
public List<T> FindAll(Predicate<T> match)
```

Как видите, метод `FindAll()` возвращает новый объект `List<T>`, который представляет подмножество данных. Также обратите внимание, что единственным параметром `FindAll()` является обобщенный делегат типа `System.Predicate<T>`, способный указывать на любой метод, который возвращает `bool` и принимает единственный параметр:

```
// Этот делегат используется методом FindAll()
// для извлечения подмножества.
public delegate bool Predicate<T>(T obj);
```

Когда вызывается `FindAll()`, каждый элемент в `List<T>` передается методу, указанному объектом `Predicate<T>`. Реализация упомянутого метода будет выполнять некоторые вычисления для проверки соответствия элемента данных заданному критерию, возвращая в результате `true` или `false`. Если метод возвращает `true`, то текущий элемент будет добавлен в новый объект `List<T>`, который представляет интересующее подмножество.

Прежде чем мы посмотрим, как лямбда-выражения могут упростить работу с методом `FindAll()`, давайте решим задачу длинным способом, используя объекты делегатов непосредственно. Добавьте в класс `Program` метод (`TraditionalDelegateSyntax()`), который взаимодействует с типом `System.Predicate<T>` для обнаружения четных чисел в списке `List<T>` целочисленных значений:

```
using System;
using System.Collections.Generic;
using LambdaExpressions;

Console.WriteLine("***** Fun with Lambdas *****\n");
TraditionalDelegateSyntax();
Console.ReadLine();

static void TraditionalDelegateSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Вызвать FindAll() с применением традиционного синтаксиса делегатов.
  Predicate<int> callback = IsEvenNumber;
  List<int> evenNumbers = list.FindAll(callback);

  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}

// Цель для делегата Predicate<>.
static bool IsEvenNumber(int i)
{
  // Это четное число?
  return (i % 2) == 0;
}
```

Здесь имеется метод (`IsEvenNumber()`), который отвечает за проверку входного целочисленного параметра на предмет четности или нечетности с применением операции получения остатка от деления (`%`) языка C#. Запуск приложения приводит к выводу на консоль чисел 20, 4, 8 и 44.

Наряду с тем, что такой традиционный подход к работе с делегатами ведет себя ожидаемым образом, `IsEvenNumber()` вызывается только при ограниченных обстоятельствах — в частности, когда вызывается метод `FindAll()`, который возлагает на

нас обязанность по полному определению метода. Если взамен использовать анонимный метод, то можно превратить это в локальную функцию и код станет значительно чище. Добавьте в класс `Program` следующий новый метод:

```
static void AnonymousMethodSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Теперь использовать анонимный метод.
  List<int> evenNumbers = list.FindAll(delegate(int i)
    { return (i % 2) == 0; } );

  // Вывести четные числа.
  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

В данном случае вместо прямого создания объекта делегата `Predicate<T>` и последующего написания отдельного метода есть возможность определить метод как анонимный. Несмотря на шаг в правильном направлении, вам по-прежнему придется применять ключевое слово `delegate` (или строго типизированный класс `Predicate<T>`) и обеспечивать точное соответствие списка параметров:

```
List<int> evenNumbers = list.FindAll(
  delegate(int i)
  {
    return (i % 2) == 0;
  }
);
```

Для еще большего упрощения вызова метода `FindAll()` могут использоваться *лямбда-выражения*. Во время применения синтаксиса лямбда-выражений вообще не приходится иметь дело с лежащим в основе объектом делегата. Взгляните на показанный далее новый метод в классе `Program`:

```
static void LambdaExpressionSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Теперь использовать лямбда-выражение C#.
  List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

  // Вывести четные числа.
  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

Обратите внимание на довольно странный оператор кода, передаваемый методу `FindAll()`, который на самом деле и представляет собой лямбда-выражение. В такой версии примера нет вообще никаких следов делегата `Predicate<T>` (или ключевого слова `delegate`, если на то пошло). Должно указываться только лямбда-выражение:

```
i => (i % 2) == 0
```

Перед разбором синтаксиса запомните, что лямбда-выражения могут использоваться везде, где должен применяться анонимный метод или строго типизированный делегат (обычно с клавиатурным набором гораздо меньшего объема). "За кулисами" компилятор C# транслирует лямбда-выражение в стандартный анонимный метод, использующий тип делегата `Predicate<T>` (в чем можно удостовериться с помощью утилиты `ildasm.exe`). Скажем, следующий оператор кода:

```
// Это лямбда-выражение...
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

компилируется в приблизительно такой код C#:

```
// ...становится следующим анонимным методом.
List<int> evenNumbers = list.FindAll(delegate (int i)
{
  return (i % 2) == 0;
});
```

Анализ лямбда-выражения

Лямбда-выражение начинается со списка параметров, за которым следует лексема => (лексема C# для лямбда-операции позаимствована из области *лямбда-исчисления*), а за ней — набор операторов (или одиночный оператор), который будет обрабатывать передаваемые аргументы. На самом высоком уровне лямбда-выражение можно представить следующим образом:

```
АргументыДляОбработки => ОбрабатывающиеОператоры
```

То, что находится внутри метода `LambdaExpressionSyntax()`, понимается так:

```
// i - список параметров.
// (i % 2) == 0 - набор операторов для обработки i.
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

Параметры лямбда-выражения могут быть явно или неявно типизированными. В настоящий момент тип данных, представляющий параметр `i` (целочисленное значение), определяется неявно. Компилятор в состоянии понять, что `i` является целочисленным значением, на основе области действия всего лямбда-выражения и лежащего в основе делегата. Тем не менее, определять тип каждого параметра в лямбда-выражении можно также и явно, помещая тип данных и имя переменной в пару круглых скобок, как показано ниже:

```
// Теперь установим тип параметров явно.
List<int> evenNumbers = list.FindAll((int i) => (i % 2) == 0);
```

Как вы уже видели, если лямбда-выражение имеет одиночный неявно типизированный параметр, то круглые скобки в списке параметров могут быть опущены. Если вы желаете соблюдать согласованность относительно применения параметров лямбда-выражений, тогда можете *всегда* заключать в скобки список параметров:

```
List<int> evenNumbers = list.FindAll((i) => (i % 2) == 0);
```

Наконец, обратите внимание, что в текущий момент выражение не заключено в круглые скобки (естественно, вычисление остатка от деления помещено в скобки, чтобы гарантировать его выполнение перед проверкой на равенство). В лямбда-выражениях разрешено заключать оператор в круглые скобки:

```
// Поместить в скобки и выражение.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

После ознакомления с разными способами построения лямбда-выражения давайте выясним, как его можно читать в понятных человеку терминах. Оставив чистую математику в стороне, можно привести следующее объяснение:

```
// Список параметров (в данном случае единственное целочисленное
// значение по имени i) будет обработан выражением (i % 2) == 0.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Обработка аргументов внутри множества операторов

Первое рассмотренное лямбда-выражение включало единственный оператор, который в итоге вычислялся в булевское значение. Однако, как вы знаете, многие цели делегатов должны выполнять несколько операторов кода. По этой причине язык C# позволяет строить лямбда-выражения, состоящие из множества операторов, указывая блок кода в стандартных фигурных скобках. Взгляните на приведенную далее модификацию метода `LambdaExpressionSyntax()`:

```
static void LambdaExpressionSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Обработать каждый аргумент внутри группы операторов кода.
  List<int> evenNumbers = list.FindAll((i) =>
  {
    Console.WriteLine("value of i is currently: {0}", i);
                  // текущее значение i
    bool isEven = ((i % 2) == 0);
    return isEven;
  });

  // Вывести четные числа.
  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

В данном случае список параметров (опять состоящий из единственного целочисленного значения `i`) обрабатывается набором операторов кода. Помимо вызова метода `Console.WriteLine()` оператор вычисления остатка от деления разбит на два оператора ради повышения читабельности. Предположим, что каждый из рассмотренных выше методов вызывается внутри операторов верхнего уровня:

```
Console.WriteLine("***** Fun with Lambdas *****\n");
TraditionalDelegateSyntax();
AnonymousMethodSyntax();
Console.WriteLine();
LambdaExpressionSyntax();
Console.ReadLine();
```

Запуск приложения дает следующий вывод:

```
***** Fun with Lambdas *****

Here are your even numbers:
20     4      8      44
Here are your even numbers:
20     4      8      44
value of i is currently: 20
value of i is currently: 1
value of i is currently: 4
value of i is currently: 8
value of i is currently: 9
value of i is currently: 44
Here are your even numbers:
20     4      8      44
```

Лямбда-выражения с несколькими параметрами и без параметров

Показанные ранее лямбда-выражения обрабатывали единственный параметр. Тем не менее, это вовсе не обязательно, т.к. лямбда-выражения могут обрабатывать множество аргументов (или ни одного). Для демонстрации первого сценария с множеством аргументов добавьте показанную ниже версию класса `SimpleMath`:

```
public class SimpleMath
{
  public delegate void MathMessage(string msg, int result);
  private MathMessage _mmDelegate;
  public void SetMathHandler(MathMessage target)
  {
    _mmDelegate = target;
  }
  public void Add(int x, int y)
  {
    _mmDelegate?.Invoke("Adding has completed!", x + y);
  }
}
```

Обратите внимание, что делегат `MathMessage` ожидает два параметра. Чтобы представить их в виде лямбда-выражения, операторы верхнего уровня можно записать так:

```
// Зарегистрировать делегат как лямбда-выражение.
SimpleMath m = new SimpleMath();
m.SetMathHandler((msg, result) =>
  {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});

// Это приведет к выполнению лямбда-выражения.
m.Add(10, 10);
Console.ReadLine();
```

Здесь задействовано выведение типа, поскольку для простоты два параметра не были строго типизированы. Однако метод SetMathHandler() можно было бы вызвать следующим образом:

```
m.SetMathHandler((string msg, int result) =>
  {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});
```

Наконец, если лямбда-выражение применяется для взаимодействия с делегатом, который вообще не принимает параметров, то можно указать в качестве параметра пару пустых круглых скобок. Таким образом, предполагая, что определен приведенный далее тип делегата:

```
public delegate string VerySimpleDelegate();
```

вот как можно было бы обработать результат вызова:

```
// Выводит на консоль строку "Enjoy your string!".
VerySimpleDelegate d =
   new VerySimpleDelegate( () => {return "Enjoy your string!";} );
Console.WriteLine(d());
```

Используя новый синтаксис выражений, предыдущую строку можно записать следующим образом:

```
VerySimpleDelegate d2 =
  new VerySimpleDelegate(() => "Enjoy your string!");
```

и даже сократить ее до такого вида:

```
VerySimpleDelegate d3 = () => "Enjoy your string!";
```

Использование ключевого слова static с лямбда-выражениями (нововведение в версии 9.0)

Поскольку лямбда-выражения являются сокращенной формой записи для делегатов, должно быть понятно, что они тоже поддерживают ключевое слово static (начиная с версии C# 9.0) и отбрасывание (рассматривается в следующем разделе). Добавьте к операторам верхнего уровня такой код:

```
var outerVariable = 0;

Func<int, int, bool> DoWork = (x,y) =>
{
  outerVariable++;
  return true;
};
DoWork(3,4);
Console.WriteLine("Outer variable now = {0}", outerVariable);
```

В результате выполнения этого кода получается следующий вывод:

```
***** Fun with Lambdas *****

Outer variable now = 1
```

Если вы сделаете лямбда-выражение статическим, тогда на этапе компиляции возникнет ошибка, т.к. выражение пытается модифицировать переменную, объявленную во внешней области действия:

```
var outerVariable = 0;
Func<int, int, bool> DoWork = static (x,y) =>
{
  // Ошибка на этапе компиляции по причине доступа
  // к внешней переменной.
  // outerVariable++;
  return true;
};
```

Использование отбрасывания с лямбда-выражениями (нововведение в версии 9.0)

Как и в случае делегатов (начиная с версии C# 9.0), входные переменные лямбда-выражения можно заменять отбрасыванием, когда они не нужны. Здесь применяется та же самая уловка, что и в делегатах. Поскольку символ подчеркивания (_) в предшествующих версиях C# считался законным идентификатором переменной, в лямбда-выражении должно присутствовать два и более подчеркиваний, чтобы они трактовались как отбрасывание:

```
var outerVariable = 0;
Func<int, int, bool> DoWork = (x,y) =>
{
  outerVariable++;
  return true;
};
DoWork(_,_);
Console.WriteLine("Outer variable now = {0}", outerVariable);
```

Модернизация примера `CarEvents` с использованием лямбда-выражений

С учетом того, что основной целью лямбда-выражений является предоставление способа ясного и компактного определения анонимных методов (косвенно упрощая работу с делегатами), давайте модернизируем проект `CarEvents`, созданный ранее в главе. Ниже приведена упрощенная версия класса `Program` из упомянутого проекта, в которой для перехвата всех событий, поступающих от объекта `Car`, применяется синтаксис лямбда-выражений (вместо простых делегатов):

```
using System;
using CarEventsWithLambdas;
Console.WriteLine("***** More Fun with Lambdas *****\n");
// Создать объект Car обычным образом.
Car c1 = new Car("SlugBug", 100, 10);

// Привязаться к событиям с помощью лямбда-выражений.
c1.AboutToBlow += (sender, e) => { Console.WriteLine(e.msg);};
c1.Exploded += (sender, e) => { Console.WriteLine(e.msg); };

// Увеличить скорость (это инициирует события).
Console.WriteLine("\n***** Speeding up *****");
for (int i = 0; i < 6; i++)
{
  c1.Accelerate(20);
}
Console.ReadLine();
```

Лямбда-выражения и члены, сжатые до выражений (обновление в версии 7.0)

Понимая лямбда-выражения и зная, как они работают, вам должно стать намного яснее, каким образом внутренне функционируют члены, сжатые до выражений. В главе 4 упоминалось, что в версии C# 6 появилась возможность использовать операцию => для упрощения некоторых реализаций членов. В частности, если есть метод или свойство (в дополнение к специальной операции или процедуре преобразования, как было показано в главе 11), реализация которого содержит единственную строку кода, тогда определять область действия посредством фигурных скобок необязательно. Взамен можно задействовать лямбда-операцию и написать член, сжатый до выражения. В версии C# 7 такой синтаксис можно применять для конструкторов и финализаторов классов (раскрываемых в главе 9), а также для средств доступа `get` и `set` к свойствам.

Тем не менее, имейте в виду, что новый сокращенный синтаксис может применяться где угодно, даже когда код не имеет никакого отношения к делегатам или событиям. Таким образом, например, если вы строите элементарный класс для сложения двух чисел, то могли бы написать следующий код:

```
class SimpleMath
{
  public int Add(int x, int y)
  {
    return x + y;
  }
  public void PrintSum(int x, int y)
  {
    Console.WriteLine(x + y);
  }
}
```

В качестве альтернативы теперь код может выглядеть так:

```
class SimpleMath
{
  public int Add(int x, int y) =>  x + y;
  public void PrintSum(int x, int y) => Console.WriteLine(x + y);
}
```

В идеале к этому моменту вы должны уловить суть лямбда-выражений и понимать, что они предлагают "функциональный способ" работы с анонимными методами и типами делегатов. Хотя на привыкание к лямбда-операции (=>) может уйти некоторое время, просто запомните, что лямбда-выражение сводится к следующей форме:

```
АргументыДляОбработки =>
{
  ОбрабатывающиеОператоры
}
```

Или, если операция => используется для реализации члена типа с единственным оператором, то это будет выглядеть так:

```
ЧленТипа => ЕдинственныйОператорКода
```

Полезно отметить, что лямбда-выражения широко задействованы также в модели программирования LINQ, помогая упростить кодирование. Исследование LINQ начинается в главе 13.

Резюме

В настоящей главе вы получили представление о нескольких способах организации двустороннего взаимодействия для множества объектов. Во-первых, было рассмотрено ключевое слово `delegate`, которое применяется для косвенного конструирования класса, производного от `System.MulticastDelegate`. Вы узнали, что объект делегата поддерживает список методов для вызова тогда, когда ему об этом будет указано.

Во-вторых, вы ознакомились с ключевым словом `event`, которое в сочетании с типом делегата может упростить процесс отправки уведомлений ожидающим объектам. Как можно заметить в результирующем коде CIL, модель событий .NET отображается на скрытые обращения к типам `System.Delegate/System.MulticastDelegate`. В данном отношении ключевое слово `event` является совершенно необязательным, т.к. оно просто позволяет сэкономить на наборе кода. Кроме того, вы видели, что `null`-условная операция C# 6.0 упрощает безопасное инициирование событий для любой заинтересованной стороны.

В-третьих, в главе также рассматривалось средство языка C#, которое называется *анонимными методами*. С помощью такой синтаксической конструкции можно явно ассоциировать блок операторов кода с заданным событием. Было показано, что анонимные методы вполне могут игнорировать параметры, переданные событием, и имеют доступ к "внешним переменным" определяющего их метода. Вы также освоили упрощенный подход к регистрации событий с применением *групповых преобразований методов*.

Наконец, в-четвертых, вы взглянули на лямбда-операцию (=>) языка C#. Как было показано, этот синтаксис представляет собой сокращенный способ для записи анонимных методов, когда набор аргументов может быть передан на обработку группе операторов. Любой метод внутри платформы .NET Core, который принимает объект делегата в качестве аргумента, может быть заменен связанным лямбда-выражением, что обычно несколько упрощает кодовую базу.

ГЛАВА 13

LINQ to Objects

Независимо от типа приложения, которое вы создаете с использованием платформы .NET Core, во время выполнения ваша программа непременно будет нуждаться в доступе к данным какой-нибудь формы. Разумеется, данные могут находиться в многочисленных местах, включая файлы XML, реляционные базы данных, коллекции в памяти и элементарные массивы. Исторически сложилось так, что в зависимости от места хранения данных программистам приходилось применять разные и несвязанные друг с другом API-интерфейсы. Набор технологий LINQ (Language Integrated Query — язык интегрированных запросов), появившийся в версии .NET 3.5, предоставил краткий, симметричный и строго типизированный способ доступа к широкому разнообразию хранилищ данных. В настоящей главе изучение LINQ начинается с исследования LINQ to Objects.

Прежде чем погрузиться в LINQ to Objects, в первой части главы предлагается обзор основных программных конструкций языка C#, которые делают возможным существование LINQ. По мере чтения главы вы убедитесь, насколько полезны (а иногда и обязательны) такие средства, как неявно типизированные переменные, синтаксис инициализации объектов, лямбда-выражения, расширяющие методы и анонимные типы.

После пересмотра поддерживающей инфраструктуры в оставшемся материале главы будет представлена модель программирования LINQ и объяснена ее роль в рамках платформы .NET. Вы узнаете, для чего предназначены операции и выражения запросов, позволяющие определять операторы, которые будут опрашивать источник данных с целью выдачи требуемого результирующего набора. Попутно будут строиться многочисленные примеры LINQ, взаимодействующие с данными в массивах и коллекциях различного типа (обобщенных и необобщенных), а также исследоваться сборки, пространства имен и типы, которые представляют API-интерфейс LINQ to Objects.

На заметку! Информация, приведенная в главе, послужит фундаментом для освоения материала последующих глав книги, включая Parallel LINQ (глава 15) и Entity Framework Core (главы 22 и 23).

Программные конструкции, специфичные для LINQ

С высокоуровневой точки зрения LINQ можно трактовать как строго типизированный язык запросов, встроенный непосредственно в грамматику самого языка C#. Используя LINQ, можно создавать любое количество выражений, которые выглядят и ведут себя подобно SQL-запросам к базе данных. Однако запрос LINQ может применяться к любым хранилищам данных, включая хранилища, которые не имеют ничего общего с подлинными реляционными базами данных.

На заметку! Хотя запросы LINQ внешне похожи на запросы SQL, их синтаксис не идентичен. В действительности многие запросы LINQ имеют формат, прямо противоположный формату подобного запроса к базе данных! Если вы попытаетесь отобразить LINQ непосредственно на SQL, то определенно запутаетесь. Чтобы подобного не произошло, рекомендуется воспринимать запросы LINQ как уникальные операторы, которые просто случайно оказались похожими на SQL.

Когда LINQ впервые был представлен в составе платформы .NET 3.5, языки C# и VB уже были расширены множеством новых программных конструкций для поддержки набора технологий LINQ. В частности, язык C# использует следующие связанные с LINQ средства:

- неявно типизированные локальные переменные;
- синтаксис инициализации объектов и коллекций;
- лямбда-выражения;
- расширяющие методы;
- анонимные типы.

Перечисленные средства уже детально рассматривались в других главах книги. Тем не менее, чтобы освежить все в памяти, давайте быстро вспомним о каждом средстве по очереди, удостоверившись в правильном их понимании.

На заметку! Из-за того, что в последующих разделах приводится обзор материала, рассматриваемого где-то в других местах книги, проект кода C# здесь не предусмотрен.

Неявная типизация локальных переменных

В главе 3 вы узнали о ключевом слове `var` языка C#. Оно позволяет определять локальную переменную без явного указания типа данных. Однако такая переменная будет строго типизированной, потому что компилятор определит ее корректный тип данных на основе начального присваивания. Вспомните показанный ниже код примера из главы 3:

```
static void DeclareImplicitVars()
{
  // Неявно типизированные локальные переменные.
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";

  // Вывести имена лежащих в основе типов.
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
  Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
  Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

Это языковое средство удобно и зачастую обязательно, когда применяется LINQ. Как вы увидите на протяжении главы, многие запросы LINQ возвращают последовательность типов данных, которые не будут известны вплоть до этапа компиляции. Учитывая, что лежащий в основе тип данных не известен до того, как приложение скомпилируется, вполне очевидно, что явно объявить такую переменную невозможно!

Синтаксис инициализации объектов и коллекций

В главе 5 объяснялась роль синтаксиса инициализации объектов, который позволяет создавать переменную типа класса или структуры и устанавливать любое количество ее открытых свойств за один прием. В результате получается компактный (и по-прежнему легко читаемый) синтаксис, который может использоваться для подготовки объектов к потреблению. Также вспомните из главы 10, что язык C# поддерживает похожий синтаксис инициализации коллекций объектов. Взгляните на следующий фрагмент кода, где синтаксис инициализации коллекций применяется для наполнения `List<T>` объектами `Rectangle`, каждый из которых состоит из пары объектов `Point`, представляющих точку с координатами (x, y):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
  new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                 BottomRight = new Point { X = 200, Y = 200}},
  new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                 BottomRight = new Point { X = 100, Y = 100}},
  new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                 BottomRight = new Point { X = 90, Y = 75}}
};
```

Несмотря на то что использовать синтаксис инициализации коллекций или объектов совершенно не обязательно, с его помощью можно получить более компактную кодовую базу. Кроме того, этот синтаксис в сочетании с неявной типизацией локальных переменных позволяет объявлять анонимный тип, что удобно при создании проекций LINQ. О проекциях LINQ речь пойдет позже в главе.

Лямбда-выражения

Лямбда-операция C# (=>) подробно рассматривалась в главе 12. Вспомните, что данная операция позволяет строить лямбда-выражение, которое может применяться в любой момент при вызове метода, требующего строго типизированный делегат в качестве аргумента. Лямбда-выражения значительно упрощают работу с делегатами, т.к. сокращают объем кода, который должен быть написан вручную. Лямбда-выражения могут быть представлены следующим образом:

```
АргументыДляОбработки =>
{
  ОбрабатывающиеОператоры
}
```

В главе 12 было показано, как взаимодействовать с методом `FindAll()` обобщенного класса `List<T>` с использованием трех разных подходов. После работы с низкоуровневым делегатом `Predicate<T>` и анонимным методом C# мы пришли к приведенной ниже (исключительно компактной) версии, в которой использовалось лямбда-выражение:

```
static void LambdaExpressionSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
```

```
  // Теперь использовать лямбда-выражение C#.
  List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

  // Вывести четные числа.
  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

Лямбда-выражения будут удобны при работе с объектной моделью, лежащей в основе LINQ. Как вы вскоре выясните, операции запросов LINQ в C# — просто сокращенная запись для вызова методов класса по имени `System.Linq.Enumerable`. Эти методы обычно всегда требуют передачи в качестве параметров делегатов (в частности, делегата `Func<>`), которые применяются для обработки данных с целью выдачи корректного результирующего набора. За счет использования лямбда-выражений можно упростить код и позволить компилятору вывести нужный делегат.

Расширяющие методы

Расширяющие методы C# позволяют оснащать существующие классы новой функциональностью без необходимости в создании подклассов. Кроме того, расширяющие методы дают возможность добавлять новую функциональность к запечатанным классам и структурам, которые в принципе не допускают построения подклассов. Вспомните из главы 11, что когда создается расширяющий метод, первый его параметр снабжается ключевым словом `this` и помечает расширяемый тип. Также вспомните, что расширяющие методы должны всегда определяться внутри статического класса, а потому объявляться с применением ключевого слова `static`. Вот пример:

```
namespace MyExtensions
{
  static class ObjectExtensions
  {
    // Определить расширяющий метод для System.Object.
    public static void DisplayDefiningAssembly(this object obj)
    {
      Console.WriteLine("{0} lives here:\n\t->{1}\n", obj.GetType().Name,
        Assembly.GetAssembly(obj.GetType()));
    }
  }
}
```

Чтобы использовать такое расширение, приложение должно импортировать пространство имен, определяющее расширение (и возможно добавить ссылку на внешнюю сборку). Затем можно приступать к написанию кода:

```
// Поскольку все типы расширяют System.Object, все
// классы и структуры могут использовать это расширение.
int myInt = 12345678;
myInt.DisplayDefiningAssembly();

System.Data.DataSet d = new System.Data.DataSet();
d.DisplayDefiningAssembly();
```

При работе с LINQ вам редко (если вообще когда-либо) потребуется вручную строить собственные расширяющие методы. Тем не менее, создавая выражения запросов LINQ, вы на самом деле будете применять многочисленные расширяющие методы, уже определенные разработчиками из Microsoft. Фактически каждая операция запроса LINQ в C# представляет собой сокращенную запись для ручного вызова лежащего в основе расширяющего метода, который обычно определен в служебном классе `System.Linq.Enumerable`.

Анонимные типы

Последним средством языка C#, описание которого здесь кратко повторяется, являются анонимные типы, рассмотренные в главе 11. Данное средство может использоваться для быстрого моделирования "формы" данных, разрешая компилятору генерировать на этапе компиляции новое определение класса, которое основано на предоставленном наборе пар "имя-значение". Вспомните, что результирующий тип составляется с применением семантики на основе значений, а каждый виртуальный метод `System.Object` будет соответствующим образом переопределен. Чтобы определить анонимный тип, понадобится объявить неявно типизированную переменную и указать форму данных с использованием синтаксиса инициализации объектов:

```
// Создать анонимный тип, состоящий из еще одного анонимного типа.
var purchaseItem = new {
  TimeBought = DateTime.Now,
  ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
  Price = 34.000};
```

Анонимные типы часто применяются в LINQ, когда необходимо проецировать в новые формы данных на лету. Например, предположим, что есть коллекция объектов `Person`, и вы хотите использовать LINQ для получения информации о возрасте и номере карточки социального страхования в каждом объекте. Применяя проецироавние LINQ, можно предоставить компилятору возможность генерации нового анонимного типа, который содержит интересующую информацию.

Роль LINQ

На этом краткий обзор средств языка C#, которые позволяют LINQ делать свою работу, завершен. Однако важно понимать, зачем вообще нужен язык LINQ. Любой разработчик программного обеспечения согласится с утверждением, что значительное время при программировании тратится на получение и манипулирование данными. Когда говорят о "данных", на ум немедленно приходит информация, хранящаяся внутри реляционных баз данных. Тем не менее, другими популярными местоположениями для данных являются документы XML или простые текстовые файлы.

Данные могут находиться в многочисленных местах помимо указанных двух распространенных хранилищ информации. Например, пусть имеется массив или обобщенный тип `List<T>`, содержащий 300 целых чисел, и требуется получить подмножество, которое удовлетворяет заданному критерию (например, только четные или нечетные числа, только простые числа, только неповторяющиеся числа больше 50). Или, возможно, при использовании API-интерфейсов рефлексии необходимо получить в массиве элементов `Type` только описания метаданных для каждого класса, производного от какого-то родительского класса. На самом деле данные находятся *повсюду*.

До появления версии .NET 3.5 взаимодействие с отдельной разновидностью данных требовало от программистов применения совершенно несходных API-интерфейсов. В табл. 13.1 описаны некоторые популярные API-интерфейсы, используемые для доступа к разнообразным типам данных (наверняка вы в состоянии привести и другие примеры).

Таблица 13.1. Способы манипулирования различными типами данных

Интересующие данные	Способ получения
Реляционные данные	`System.Data.dll`, `System.Data.SqlClient.dll` и т.д.
Данные документов XML	`System.Xml.dll`
Таблицы метаданных	Пространство имен `System.Reflection`
Коллекции объектов	Пространства имен `System.Array` и `System.Collections/System.Collections.Generic`

Разумеется, с такими подходами к манипулированию данными не связано ничего плохого. В сущности, вы можете (и будете) работать напрямую с ADO.NET, пространствами имен XML, службами рефлексии и разнообразными типами коллекций. Однако основная проблема заключается в том, что каждый из API-интерфейсов подобного рода является "самостоятельным островком", трудно интегрируемым с другими. Правда, можно (например) сохранить объект `DataSet` из ADO.NET в документ XML и затем манипулировать им посредством пространств имен `System.Xml`, но все равно манипуляции данными остаются довольно асимметричными.

В рамках API-интерфейса LINQ была предпринята попытка предложить программистам согласованный, симметричный способ получения и манипулирования "данными" в широком смысле этого понятия. Применяя LINQ, прямо внутри языка программирования C# можно создавать конструкции, которые называются *выражениями запросов*. Такие выражения запросов основаны на многочисленных операциях запросов, которые намеренно сделаны похожими внешне и по поведению (но не идентичными) на выражения SQL.

Тем не менее, трюк заключается в том, что выражение запроса может использоваться для взаимодействия с разнообразными типами данных — даже с теми, которые не имеют ничего общего с реляционными базами данных. Строго говоря, LINQ представляет собой термин, в целом описывающий сам подход доступа к данным. Однако в зависимости от того, где применяются запросы LINQ, вы встретите разные обозначения вроде перечисленных ниже.

- *LINQ to Objects*. Этот термин относится к действию по применению запросов LINQ к массивам и коллекциям.
- *LINQ to XML*. Этот термин относится к действию по использованию LINQ для манипулирования и запрашивания документов XML.
- *LINQ to Entities*. Этот аспект LINQ позволяет использовать запросы LINQ внутри API-интерфейса ADO.NET Entity Framework (EF) Core.
- *Parallel LINQ* (*PLINQ*). Этот аспект делает возможной параллельную обработку данных, возвращаемых из запроса LINQ.

В настоящее время LINQ является неотъемлемой частью библиотек базовых классов .NET Core, управляемых языков и самой среды Visual Studio.

Выражения LINQ строго типизированы

Важно также отметить, что выражение запроса LINQ (в отличие от традиционного оператора SQL) *строго типизировано*. Следовательно, компилятор C# следит за этим и гарантирует, что выражения оформлены корректно с точки зрения синтаксиса. Инструменты вроде Visual Studio могут применять метаданные для поддержки удобных средств, таких как IntelliSense, автозавершение и т.д.

Основные сборки LINQ

Для работы с LINQ to Objects вы должны обеспечить импортирование пространства имен `System.Linq` в каждом файле кода C#, который содержит запросы LINQ. В противном случае возникнут проблемы. Удостоверьтесь, что в каждом файле кода, где используется LINQ, присутствует следующий оператор `using`:

```
using System.Linq;
```

Применение запросов LINQ к элементарным массивам

Чтобы начать исследование LINQ to Objects, давайте построим приложение, которое будет применять запросы LINQ к разнообразным объектам типа массива. Создайте новый проект консольного приложения под названием `LinqOverArray` и определите в классе `Program` статический вспомогательный метод по имени `QueryOverStrings()`. Внутри метода создайте массив типа `string`, содержащий несколько произвольных элементов (скажем, названий видеоигр). Удостоверьтесь в том, что хотя бы два элемента содержат числовые значения и несколько элементов включают внутренние пробелы:

```
static void QueryOverStrings()
{
  // Предположим, что есть массив строк.
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                "Fallout 3", "Daxter", "System Shock 2"};
}
```

Теперь модифицируйте файл `Program.cs` с целью вызова метода `QueryOverStrings()`:

```
Console.WriteLine("***** Fun with LINQ to Objects *****\n");
QueryOverStrings();
Console.ReadLine();
```

При работе с любым массивом данных часто приходится извлекать из него подмножество элементов на основе определенного критерия. Возможно, требуется получить только элементы, которые содержат число (например, "System Shock 2", "Uncharted 2" и "Fallout 3"), содержат заданное количество символов либо не содержат встроенных пробелов (скажем, "Morrowind" или "Daxter"). В то время как такие задачи определенно можно решать с использованием членов типа `System.Array`, прикладывая приличные усилия, выражения запросов LINQ значительно упрощают процесс.

Исходя из предположения, что из массива нужно получить только элементы, содержащие внутри себя пробел, и представить их в алфавитном порядке, можно построить следующее выражение запроса LINQ:

```
static void QueryOverStrings()
{
  // Предположим, что имеется массив строк.
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // Построить выражение запроса для нахождения
  // элементов массива, которые содержат пробелы.
  IEnumerable<string> subset =
    from g in currentVideoGames
    where g.Contains(" ")
    orderby g
    select g;

  // Вывести результаты.
  foreach (string s in subset)
  {
    Console.WriteLine("Item: {0}", s);
  }
}
```

Обратите внимание, что в созданном здесь выражении запроса применяются операции `from`, `in`, `where`, `orderby` и `select` языка LINQ. Формальности синтаксиса выражений запросов будут подробно излагаться далее в главе. Тем не менее, даже сейчас вы в состоянии прочесть данный оператор примерно так: "предоставить мне элементы из `currentVideoGames`, содержащие пробелы, в алфавитном порядке".

Каждому элементу, который соответствует критерию поиска, назначается имя `g` (от "game"), но подошло бы любое допустимое имя переменной C#:

```
IEnumerable<string> subset =
  from game in currentVideoGames
  where game.Contains(" ")
  orderby game
  select game;
```

Возвращенная последовательность сохраняется в переменной по имени `subset`, которая имеет тип, реализующий обобщенную версию интерфейса `IEnumerable<T>`, где `T` — тип `System.String` (в конце концов, вы запрашиваете массив элементов `string`). После получения результирующего набора его элементы затем просто выводятся на консоль с использованием стандартной конструкции `foreach`. Запустив приложение, вы получите следующий вывод:

```
***** Fun with LINQ to Objects *****
Item: Fallout 3
Item: System Shock 2
Item: Uncharted 2
```

Решение с использованием расширяющих методов

Применяемый ранее (и далее в главе) синтаксис LINQ называется *выражениями запросов* LINQ, которые представляют собой формат, похожий на SQL, но слегка отличающийся от него. Существует еще один синтаксис с расширяющими методами, который будет использоваться в большинстве примеров в настоящей книге.

Создайте новый метод по имени `QueryOverStringsWithExtensionMethods()` со следующим кодом:

```
static void QueryOverStringsWithExtensionMethods()
{
  // Пусть имеется массив строк.
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                "Fallout 3", "Daxter",
                                "System Shock 2"};

  // Построить выражение запроса для поиска
  // в массиве элементов, содержащих пробелы.
  IEnumerable<string> subset =
    currentVideoGames.Where(g => g.Contains(" "))
                     .OrderBy(g => g).Select(g => g);

  // Вывести результаты.
  foreach (string s in subset)
  {
    Console.WriteLine("Item: {0}", s);
  }
}
```

Код здесь тот же, что и в предыдущем методе, кроме строк, выделенных полужирным. В них демонстрируется применение синтаксиса расширяющих методов, в котором для определения операций внутри каждого метода используются лямбда-выражения. Например, лямбда-выражение в методе `Where()` определяет условие (содержит ли значение пробел). Как и в синтаксисе выражений запросов, используемая для идентификации значения буква произвольна; в примере применяется v для видеоигр (video game).

Хотя результаты аналогичны (метод дает такой же вывод, как и предыдущий метод, использующий выражение запроса), вскоре вы увидите, что *тип* результирующего набора несколько отличается. В большинстве (если фактически не во всех) сценариях подобное отличие не приводит к каким-либо проблемам и форматы могут применяться взаимозаменяемо.

Решение без использования LINQ

Конечно, применение LINQ никогда не бывает обязательным. При желании идентичный результирующий набор можно получить без участия LINQ с помощью таких программных конструкций, как операторы `if` и циклы `for`. Ниже приведен метод, который выдает тот же самый результат, что и `QueryOverStrings()`, но в намного более многословной манере:

```
static void QueryOverStringsLongHand()
{
  // Предположим, что имеется массив строк.
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  string[] gamesWithSpaces = new string[5];

  for (int i = 0; i < currentVideoGames.Length; i++)
  {
    if (currentVideoGames[i].Contains(" "))
    {
      gamesWithSpaces[i] = currentVideoGames[i];
    }
  }
```

```
  // Отсортировать набор.
  Array.Sort(gamesWithSpaces);

  // Вывести результаты.
  foreach (string s in gamesWithSpaces)
  {
    if (s != null)
    {
      Console.WriteLine("Item: {0}", s);
    }
  }
  Console.WriteLine();
}
```

Несмотря на возможные пути улучшения метода `QueryOverStringsLongHand()`, факт остается фактом — запросы LINQ способны радикально упростить процесс извлечения новых подмножеств данных из источника. Вместо построения вложенных циклов, сложной логики `if/else`, временных типов данных и т.п. компилятор C# сделает всю черновую работу, как только вы создадите подходящий запрос LINQ.

Выполнение рефлексии результирующего набора LINQ

А теперь определите в классе `Program` дополнительный вспомогательный метод по имени `ReflectOverQueryResults()`, который выводит на консоль разнообразные детали о результирующем наборе LINQ (обратите внимание на параметр типа `System.Object`, позволяющий учитывать множество типов результирующих наборов):

```
static void ReflectOverQueryResults(object resultSet,
                                    string queryType = "Query Expressions")
{
  Console.WriteLine($"***** Info about your query using {queryType} *****");
  // Вывести тип результирующего набора.
  Console.WriteLine("resultSet is of type: {0}", resultSet.GetType().Name);
  // Вывести местоположение результирующего набора.
  Console.WriteLine("resultSet location: {0}",
                    resultSet.GetType().Assembly.GetName().Name);
}
```

Модифицируйте код метода `QueryOverStrings()` следующим образом:

```
// Построить выражение запроса для поиска
// в массиве элементов, содержащих пробел.
IEnumerable<string> subset = from g in currentVideoGames
                             where g.Contains(" ") orderby g select g;

ReflectOverQueryResults(subset);

// Вывести результаты.
foreach (string s in subset)
{
  Console.WriteLine("Item: {0}", s);
}
```

Запустив приложение, легко заметить, что переменная `subset` в действительности представляет собой экземпляр обобщенного типа `OrderedEnumerable<TElement, TKey>` (представленного в коде CIL как `OrderedEnumerable`2`), который является внутренним абстрактным типом, находящимся в сборке `System.Linq.dll`:

```
***** Info about your query using Query Expressions*****
resultSet is of type: OrderedEnumerable`2
resultSet location: System.Linq
```

Внесите такое же изменение в код метода `QueryOverStringsWithExtensionMethods()`, но передав во втором параметре строку `"Extension Methods"`:

```
// Построить выражение запроса для поиска
// в массиве элементов, содержащих пробел.
IEnumerable<string> subset =
currentVideoGames
  .Where(g => g.Contains(" "))
  .OrderBy(g => g)
  .Select(g => g);

ReflectOverQueryResults(subset,"Extension Methods");

// Вывести результаты.
foreach (string s in subset)
{
  Console.WriteLine("Item: {0}", s);
}
```

После запуска приложения выяснится, что переменная `subset` является экземпляром типа `SelectIPartitionIterator`. Но если удалить из запроса конструкцию `Select(g=>g)`, то `subset` снова станет экземпляром типа `OrderedEnumerable<TElement, TKey>`. Что все это значит? Для подавляющего большинства разработчиков немногое (если вообще что-либо). Оба типа являются производными от `IEnumerable<T>`, проход по ним осуществляется одинаковым образом и они оба способны создавать список или массив своих значений.

```
***** Info about your query using Extension Methods *****
resultSet is of type: SelectIPartitionIterator`2
resultSet location: System.Linq
```

LINQ и неявно типизированные локальные переменные

Хотя в приведенной программе относительно легко выяснить, что результирующий набор может быть интерпретирован как перечисление объектов `string` (например, `IEnumerable<string>`), тот факт, что подмножество на самом деле имеет тип `OrderedEnumerable<TElement, TKey>`, *не* настолько ясен.

Поскольку результирующие наборы LINQ могут быть представлены с применением порядочного количества типов из разнообразных пространств имен LINQ, было бы утомительно определять подходящий тип для хранения результирующего набора. Причина в том, что во многих случаях лежащий в основе тип не очевиден и даже напрямую не доступен в коде (и как вы увидите, в ряде ситуаций тип генерируется на этапе компиляции).

Чтобы еще больше подчеркнуть данное обстоятельство, ниже показан дополнительный вспомогательный метод, определенный внутри класса `Program`:

```
static void QueryOverInts()
{
  int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};

  // Вывести только элементы меньше 10.
  IEnumerable<int> subset = from i in numbers where i < 10 select i;
```

```
  foreach (int i in subset)
  {
    Console.WriteLine("Item: {0}", i);
  }
  ReflectOverQueryResults(subset);
}
```

В рассматриваемом случае переменная `subset` имеет совершенно другой внутренний тип. На этот раз тип, реализующий интерфейс `IEnumerable<int>`, представляет собой низкоуровневый класс по имени `WhereArrayIterator<T>`:

```
Item: 1
Item: 2
Item: 3
Item: 8
***** Info about your query *****
resultSet is of type: WhereArrayIterator`1
resultSet location: System.Linq
```

Учитывая, что точный тип запроса LINQ не вполне очевиден, в первых примерах результаты запросов были представлены как переменная `IEnumerable<T>`, где `T` — тип данных в возвращенной последовательности (`string`, `int` и т.д.). Тем не менее, ситуация по-прежнему довольно запутана. Чтобы еще больше все усложнить, стоит упомянуть, что поскольку интерфейс `IEnumerable<T>` расширяет необобщенный `IEnumerable`, получать результат запроса LINQ допускается и так:

```
System.Collections.IEnumerable subset =
  from i in numbers
  where i < 10
  select i;
```

К счастью, неявная типизация при работе с запросами LINQ значительно проясняет картину:

```
static void QueryOverInts()
{
  int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
  // Здесь используется неявная типизация...
  var subset = from i in numbers where i < 10 select i;
  // ...и здесь тоже.
  foreach (var i in subset)
  {
    Console.WriteLine("Item: {0} ", i);
  }
  ReflectOverQueryResults(subset);
}
```

В качестве эмпирического правила: при захвате результатов запроса LINQ всегда необходимо использовать неявную типизацию. Однако помните, что (в большинстве случаев) *действительное* возвращаемое значение имеет тип, реализующий интерфейс `IEnumerable<T>`.

Какой точно тип кроется за ним (`OrderedEnumerable<TElement, TKey>`, `WhereArrayIterator<T>` и т.п.), к делу не относится, и определять его вовсе не обязательно. Как было показано в предыдущем примере кода, для прохода по извлеченным данным можно просто применить ключевое слово `var` внутри конструкции `foreach`.

LINQ и расширяющие методы

Несмотря на то что в текущем примере совершенно не требуется напрямую писать какие-то расширяющие методы, на самом деле они благополучно используются на заднем плане. Выражения запросов LINQ могут применяться для прохода по содержимому контейнеров данных, которые реализуют обобщенный интерфейс `IEnumerable<T>`. Тем не менее, класс `System.Array` (используемый для представления массива строк и массива целых чисел) *не* реализует этот контракт:

```
// Похоже, что тип System.Array не реализует
// корректную инфраструктуру для выражений запросов!
public abstract class Array : ICloneable, IList,
    IStructuralComparable, IStructuralEquatable
{
  ...
}
```

Хотя класс `System.Array` не реализует напрямую интерфейс `IEnumerable<T>`, он косвенно получает необходимую функциональность данного типа (а также многие другие члены, связанные с LINQ) через статический тип класса `System.Linq.Enumerable`.

В служебном классе `System.Linq.Enumerable` определено множество обобщенных расширяющих методов (таких как `Aggregate<T>()`, `First<T>()`, `Max<T>()` и т.д.), которые класс `System.Array` (и другие типы) получают в свое распоряжение на заднем плане. Таким образом, если вы примените операцию точки к локальной переменной `currentVideoGames`, то обнаружите большое количество членов, которые *отсутствуют* в формальном определении `System.Array`.

Роль отложенного выполнения

Еще один важный момент, касающийся выражений запросов LINQ, заключается в том, что фактически они не оцениваются до тех пор, пока не начнется итерация по результирующей последовательности. Формально это называется *отложенным выполнением*. Преимущество такого подхода связано с возможностью применения одного и того же запроса LINQ многократно к тому же самому контейнеру и полной гарантией получения актуальных результатов. Взгляните на следующее обновление метода `QueryOverInts()`:

```
static void QueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // Получить числа меньше 10.
  var subset = from i in numbers where i < 10 select i;

  // Оператор LINQ здесь оценивается!
  foreach (var i in subset)
  {
    Console.WriteLine("{0} < 10", i);
  }
  Console.WriteLine();
  // Изменить некоторые данные в массиве.
  numbers[0] = 4;
```

```
    // Снова производится оценка!
    foreach (var j in subset)
    {
      Console.WriteLine("{0} < 10", j);
    }
    Console.WriteLine();
    ReflectOverQueryResults(subset);
  }
```

На заметку! Когда оператор LINQ выбирает одиночный элемент (с использованием `First()`/`FirstOrDefault()`, `Single()`/`SingleOrDefault()` или любого метода агрегирования), запрос выполняется немедленно. Методы `First()`, `FirstOrDefault()`, `Single()` и `SingleOrDefault` будут описаны в следующем разделе. Методы агрегирования раскрываются позже в главе.

Ниже показан вывод, полученный в результате запуска программы. Обратите внимание, что во второй итерации по запрошенной последовательности появился дополнительный член, т.к. для первого элемента массива было установлено значение меньше 10:

```
1 < 10
2 < 10
3 < 10
8 < 10

4 < 10
1 < 10
2 < 10
3 < 10
8 < 10
```

Среда Visual Studio обладает одной полезной особенностью: если вы поместите точку останова перед оценкой запроса LINQ, то получите возможность просматривать содержимое во время сеанса отладки. Просто наведите курсор мыши на переменную результирующего набора LINQ (`subset` на рис. 13.1) и вам будет предложено выполнить запрос, развернув узел Results View (Представление результатов).

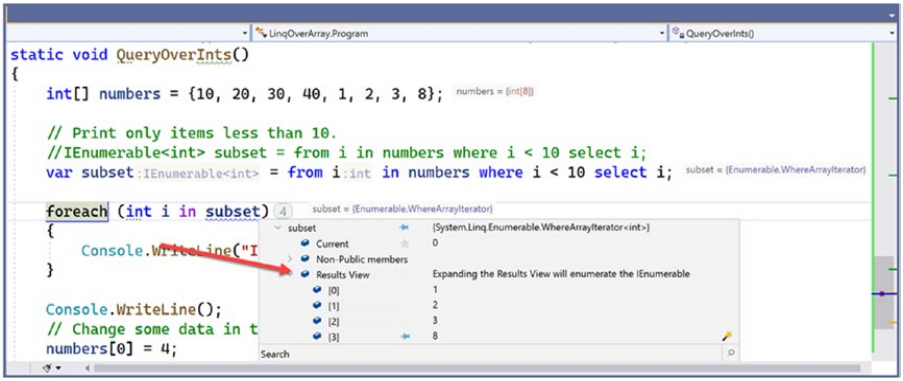

Рис. 13.1. Отладка выражений LINQ

Роль немедленного выполнения

Когда требуется оценить выражение LINQ, выдающее последовательность, за пределами логики `foreach`, можно вызывать любое количество расширяющих методов, определенных в типе `Enumerable`, таких как `ToArray<T>()`, `ToDictionary<TSource,TKey>()` и `ToList<T>()`. Все методы приводят к выполнению запроса LINQ в момент их вызова для получения снимка данных. Затем полученным снимком данных можно манипулировать независимым образом.

Кроме того, запрос выполняется немедленно в случае поиска только одного элемента. Метод `First()` возвращает первый элемент последовательности (и должен всегда применяться с конструкцией `orderby`). Метод `FirstOrDefault()` возвращает стандартное значение для типа элемента в последовательности, если возвращать нечего, например, когда исходная последовательность пуста или конструкция `where` отбросила все элементы. Метод `Single()` также возвращает первый элемент последовательности (на основе `orderby` или согласно порядку следования элементов, если конструкция `orderby` отсутствует). Подобно аналогично именованному эквиваленту метод `SingleOrDefault()` возвращает стандартное значение для типа элемента в последовательности, если последовательность пуста (или конструкция `where` отбросила все элементы). Отличие между методами `First()` (`FirstOrDefault()`) и `Single()` (`SingleOrDefault()`) заключается в том, что `Single()` (`SingleOrDefault()`) сгенерирует исключение, если из запроса будет возвращено более одного элемента.

```
static void ImmediateExecution()
{
  Console.WriteLine();
  Console.WriteLine("Immediate Execution");
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // Получить первый элемент в порядке последовательности.
  int number = (from i in numbers select i).First();
  Console.WriteLine("First is {0}", number);

  // Получить первый элемент в порядке запроса.
  number = (from i in numbers orderby i select i).First();
  Console.WriteLine("First is {0}", number);

  // Получить один элемент, который соответствует запросу.
  number = (from i in numbers where i > 30 select i).Single();
  Console.WriteLine("Single is {0}", number);
  try
  {
    // В случае возвращения более одного элемента генерируется исключение.
    number = (from i in numbers where i > 10 select i).Single();
  }
  catch (Exception ex)
  {
    Console.WriteLine("An exception occurred: {0}", ex.Message);
  }
  // Получить данные НЕМЕДЛЕННО как int[].
  int[] subsetAsIntArray =
    (from i in numbers where i < 10 select i).ToArray<int>();
  // Получить данные НЕМЕДЛЕННО как List<int>.
  List<int> subsetAsListOfInts =
    (from i in numbers where i < 10 select i).ToList<int>();
}
```

Обратите внимание, что для вызова методов Enumerable выражение LINQ целиком помещено в круглые скобки с целью приведения к корректному внутреннему типу (каким бы он ни был).

Вспомните из главы 10, что если компилятор C# в состоянии однозначно определить параметр типа обобщенного элемента, то вы не обязаны указывать этот параметр типа. Следовательно, ToArray<T>() (или ToList<T>()) можно было бы вызвать так:

```
int[] subsetAsIntArray =
  (from i in numbers where i < 10 select i).ToArray();
```

Полезность немедленного выполнения очевидна, когда нужно возвратить результаты запроса LINQ внешнему вызывающему коду, что и будет темой следующего раздела главы.

Возвращение результатов запроса LINQ

Внутри класса (или структуры) можно определить поле, значением которого будет результат запроса LINQ. Однако для этого нельзя использовать неявную типизацию (т.к. ключевое слово var не может применяться к полям), и целью запроса LINQ не могут быть данные уровня экземпляра, а потому он должен быть статическим. С учетом указанных ограничений необходимость в написании кода следующего вида будет возникать редко:

```
class LINQBasedFieldsAreClunky
{
  private static string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // Здесь нельзя использовать неявную типизацию!
  // Тип subset должен быть известен!
  private IEnumerable<string> subset = from g in currentVideoGames
    where g.Contains(" ") orderby g select g;

  public void PrintGames()
  {
    foreach (var item in subset)
    {
      Console.WriteLine(item);
    }
  }
}
```

Запросы LINQ часто определяются внутри области действия метода или свойства. Кроме того, для упрощения программирования результирующий набор будет храниться в неявно типизированной локальной переменной, использующей ключевое слово var. Вспомните из главы 3, что неявно типизированные переменные не могут применяться для определения параметров, возвращаемых значений, а также полей класса или структуры.

Итак, вполне вероятно, вас интересует, каким образом возвратить результат запроса внешнему коду. Ответ: в зависимости от обстоятельств. Если у вас есть результирующий набор, состоящий из строго типизированных данных, такой как массив строк или список List<T> объектов Car, тогда вы могли бы отказаться от использования ключевого слова var и указать подходящий тип IEnumerable<T> либо IEnumerable

(т.к. IEnumerable<T> расширяет IEnumerable). Ниже приведен пример класса Program в новом проекте консольного приложения по имени LinqRetValues:

```
using System;
using System.Collections.Generic;
using System.Linq;

Console.WriteLine("***** LINQ Return Values *****\n");
IEnumerable<string> subset = GetStringSubset();

foreach (string item in subset)
{
  Console.WriteLine(item);
}

Console.ReadLine();

static IEnumerable<string> GetStringSubset()
{
  string[] colors = {"Light Red", "Green",
    "Yellow", "Dark Red", "Red", "Purple"};

  // Обратите внимание, что subset является
  // совместимым с IEnumerable<string> объектом.
  IEnumerable<string> theRedColors =
     from c in colors where c.Contains("Red") select c;
  return theRedColors;
}
```

Результат выглядит вполне ожидаемо:

```
Light Red
Dark Red
Red
```

Возвращение результатов LINQ посредством немедленного выполнения

Рассмотренный пример работает ожидаемым образом только потому, что возвращаемое значение GetStringSubset() и запрос LINQ внутри этого метода были строго типизированными. Если применить ключевое слово var для определения переменной subset, то возвращать значение будет разрешено, *только* если метод по-прежнему прототипирован с возвращаемым типом IEnumerable<string> (и если неявно типизированная локальная переменная на самом деле совместима с указанным возвращаемым типом).

Поскольку оперировать с типом IEnumerable<T> несколько неудобно, можно задействовать немедленное выполнение. Скажем, вместо возвращения IEnumerable<string> можно было бы возвратить просто string[] при условии трансформации последовательности в строго типизированный массив. Именно такое действие выполняет новый метод класса Program:

```
static string[] GetStringSubsetAsArray()
{
  string[] colors = {"Light Red", "Green",
    "Yellow", "Dark Red", "Red", "Purple"};

  var theRedColors = from c in colors
    where c.Contains("Red") select c;
```

```
  // Отобразить результаты в массив.
  return theRedColors.ToArray();
}
```

В таком случае вызывающий код совершенно не знает, что полученный им результат поступил от запроса LINQ, и просто работает с массивом строк вполне ожидаемым образом. Вот пример:

```
foreach (string item in GetStringSubsetAsArray())
{
  Console.WriteLine(item);
}
```

Немедленное выполнение также важно при попытке возвратить вызывающему коду результаты проецирования LINQ. Мы исследуем эту тему чуть позже в главе. А сейчас давайте посмотрим, как применять запросы LINQ к обобщенным и необобщенным объектам коллекций.

Применение запросов LINQ к объектам коллекций

Помимо извлечения результатов из простого массива данных выражения запросов LINQ могут также манипулировать данными внутри классов из пространства имен `System.Collections.Generic`, таких как `List<T>`. Создайте новый проект консольного приложения по имени `ListOverCollections` и определите базовый класс `Car`, который поддерживает текущую скорость, цвет, производителя и дружественное имя:

```
namespace LinqOverCollections
{
  class Car
  {
    public string PetName {get; set;} = "";
    public string Color {get; set;} = "";
    public int Speed {get; set;}
    public string Make {get; set;} = "";
  }
}
```

Теперь определите внутри операторов верхнего уровня локальную переменную типа `List<T>` для хранения элементов типа `Car` и с помощью синтаксиса инициализации объектов заполните список несколькими новыми объектами `Car`:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using LinqOverCollections;
Console.WriteLine("***** LINQ over Generic Collections *****\n");
// Создать список List<> объектов Car.
List<Car> myCars = new List<Car>() {
  new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
  new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
  new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
  new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
  new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
};
Console.ReadLine();
```

Доступ к содержащимся в контейнере подобъектам

Применение запроса LINQ к обобщенному контейнеру ничем не отличается от такого же действия в отношении простого массива, потому что LINQ to Objects может использоваться с любым типом, реализующим интерфейс `IEnumerable<T>`. На этот раз цель заключается в построении выражения запроса для выборки из списка `myCars` только тех объектов `Car`, у которых значение скорости больше `55`.

После получения подмножества на консоль будет выведено имя каждого объекта `Car` за счет обращения к его свойству `PetName`. Предположим, что определен следующий вспомогательный метод (принимающий параметр `List<Car>`), который вызывается в операторах верхнего уровня:

```
static void GetFastCars(List<Car> myCars)
{
  // Найти в List<> все объекты Car, у которых значение Speed больше 55.
  var fastCars = from c in myCars where c.Speed > 55 select c;

  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

Обратите внимание, что выражение запроса захватывает из `List<T>` только те элементы, у которых значение `Speed` больше `55`. Запустив приложение, вы увидите, что критерию поиска отвечают только два элемента — `Henry` и `Daisy`.

Чтобы построить более сложный запрос, можно искать только автомобили марки BMW со значением `Speed` больше `90`. Для этого нужно просто создать составной булевский оператор с применением операции `&&` языка C#:

```
static void GetFastBMWs(List<Car> myCars)
{
  // Найти быстрые автомобили BMW!
  var fastCars = from c in myCars
                 where c.Speed > 90 && c.Make == "BMW" select c;
  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

Теперь выводится только одно имя `Henry`.

Применение запросов LINQ к необобщенным коллекциям

Вспомните, что операции запросов LINQ спроектированы для работы с любым типом, реализующим интерфейс `IEnumerable<T>` (как напрямую, так и через расширяющие методы). Учитывая то, что класс `System.Array` оснащен всей необходимой инфраструктурой, может оказаться сюрпризом, что унаследованные (необобщенные) контейнеры в пространстве имен `System.Collections` такой поддержкой не обладают. К счастью, итерация по данным, содержащимся внутри необобщенных коллекций, по-прежнему возможна с использованием обобщенного расширяющего метода `Enumerable.OfType<T>()`.

При вызове метода OfType<T>() на объекте необобщенной коллекции (наподобие ArrayList) нужно просто указать тип элемента внутри контейнера, чтобы извлечь совместимый с IEnumerable<T> объект. Сохранить этот элемент данных в коде можно посредством неявно типизированной переменной.

Взгляните на показанный ниже новый метод, который заполняет ArrayList набором объектов Car (не забудьте импортировать пространство имен System.Collections в начале файла Program.cs):

```
static void LINQOverArrayList()
{
  Console.WriteLine("***** LINQ over ArrayList *****");
  // Необобщенная коллекция объектов Car.
  ArrayList myCars = new ArrayList() {
    new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
    new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
    new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
    new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
    new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
  };
  // Трансформировать ArrayList в тип, совместимый с IEnumerable<T>.
  var myCarsEnum = myCars.OfType<Car>();
  // Создать выражение запроса, нацеленное на совместимый с IEnumerable<T> тип
  var fastCars = from c in myCarsEnum where c.Speed > 55 select c;
  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

Аналогично предшествующим примерам этот метод, вызванный в операторах верхнего уровня, отобразит только имена Henry и Daisy, основываясь на формате запроса LINQ.

Фильтрация данных с использованием метода OfType<T>()

Как вы уже знаете, необобщенные типы способны содержать любые комбинации элементов, поскольку члены этих контейнеров (вроде ArrayList) прототипированы для приема System.Object. Например, предположим, что ArrayList содержит разные элементы, часть которых являются числовыми. Получить подмножество, состоящее только из числовых данных, можно с помощью метода OfType<T>(), т.к. во время итерации он отфильтрует элементы, тип которых отличается от заданного:

```
static void OfTypeAsFilter()
{
  // Извлечь из ArrayList целочисленные значения.
  ArrayList myStuff = new ArrayList();
  myStuff.AddRange(new object[]{10,400,8,false,new Car(),"string data"});
  var myInts = myStuff.OfType<int>();
  // Выводит 10, 400 и 8.
  foreach (int i in myInts)
  {
    Console.WriteLine("Int value: {0}", i);
  }
}
```

К настоящему моменту вы уже умеете применять запросы LINQ к массивам, а также обобщенным и необобщенным коллекциям. Контейнеры подобного рода содержат элементарные типы C# (целочисленные и строковые данные) и специальные классы. Следующей задачей будет изучение многочисленных дополнительных операций LINQ, которые могут использоваться для построения более сложных и полезных запросов.

Исследование операций запросов LINQ

В языке C# предопределено порядочное число операций запросов. Некоторые часто применяемые из них перечислены в табл. 13.2. В дополнение к неполному списку операций, приведенному в табл. 13.3, класс `System.Linq.Enumerable` предлагает набор методов, которые не имеют прямого сокращенного обозначения в виде операций запросов C#, а доступны как расширяющие методы. Эти обобщенные методы можно вызывать для трансформации результирующего набора разными способами (`Reverse<>()`, `ToArray<>()`, `ToList<>()` и т.д.). Некоторые из них применяются для извлечения одиночных элементов из результирующего набора, другие выполняют разнообразные операции над множествами (`Distinct<>()`, `Union<>()`, `Intersect<>()` и т.п.), а есть еще те, что агрегируют результаты (`Count<>()`, `Sum<>()`, `Min<>()`, `Max<>()` и т.д.).

Таблица 13.2. Распространенные операции запросов LINQ

Операции запросов	Описание
`from, in`	Используются для определения основы любого выражения LINQ, позволяющей извлекать подмножество данных из подходящего контейнера
`where`	Применяется для определения ограничений относительно того, какие элементы должны извлекаться из контейнера
`select`	Используется для выборки последовательности из контейнера
`join, on, equals, into`	Выполняют соединения на основе указанного ключа. Помните, что эти "соединения" ничего не обязаны делать с данными в реляционной базе данных
`orderby, ascending, descending`	Позволяют упорядочить результирующий набор по возрастанию или убыванию
`groupby`	Выдают подмножество с данными, сгруппированными по указанному значению

Чтобы приступить к исследованию более замысловатых запросов LINQ, создайте новый проект консольного приложения по имени `FunWithLinqExpressions` и затем определите массив или коллекцию некоторых выборочных данных. В проекте `FunWithLinqExpressions` вы будете создавать массив объектов типа `ProductInfo`, определенного следующим образом:

```
namespace FunWithLinqExpressions
{
  class ProductInfo
  {
    public string Name {get; set;} = "";
    public string Description {get; set;} = "";
    public int NumberInStock {get; set;} = 0;
```

```
    public override string ToString()
      => $"Name={Name}, Description={Description}, Number in
Stock={NumberInStock}";
  }
}
```

Теперь заполните массив объектами `ProductInfo` в вызывающем коде:

```
Console.WriteLine("***** Fun with Query Expressions *****\n");

// Этот массив будет основой для тестирования...
ProductInfo[] itemsInStock = new[] {
  new ProductInfo{ Name = "Mac's Coffee",
    Description = "Coffee with TEETH", NumberInStock = 24},
  new ProductInfo{ Name = "Milk Maid Milk",
    Description = "Milk cow's love", NumberInStock = 100},
  new ProductInfo{ Name = "Pure Silk Tofu",
    Description = "Bland as Possible", NumberInStock = 120},
  new ProductInfo{ Name = "Crunchy Pops",
    Description = "Cheezy, peppery goodness", NumberInStock = 2},
  new ProductInfo{ Name = "RipOff Water",
    Description = "From the tap to your wallet", NumberInStock = 100},
  new ProductInfo{ Name = "Classic Valpo Pizza",
    Description = "Everyone loves pizza!", NumberInStock = 73}
};
// Здесь мы будем вызывать разнообразные методы!
Console.ReadLine();
```

Базовый синтаксис выборки

Поскольку синтаксическая корректность выражения запроса LINQ проверяется на этапе компиляции, вы должны помнить, что порядок следования операций критически важен. В простейшем виде каждый запрос LINQ строится с использованием операций `from`, `in` и `select`. Вот базовый шаблон, который нужно соблюдать:

```
var результат =
  from сопоставляемыйЭлемент in контейнер
  select сопоставляемыйЭлемент;
```

Элемент после операции `from` представляет элемент, соответствующий критерию запроса LINQ; именовать его можно по своему усмотрению. Элемент после операции `in` представляет контейнер данных, в котором производится поиск (массив, коллекция, документ XML и т.д.).

Рассмотрим простой запрос, не делающий ничего кроме извлечения каждого элемента контейнера (по поведению похожий на SQL-оператор `SELECT *` в базе данных):

```
static void SelectEverything(ProductInfo[] products)
{
  // Получить все!
  Console.WriteLine("All product details:");
  var allProducts = from p in products select p;
  foreach (var prod in allProducts)
  {
    Console.WriteLine(prod.ToString());
  }
}
```

По правде говоря, это выражение запроса не особенно полезно, т.к. оно выдает подмножество, идентичное содержимому входного параметра. При желании можно извлечь только значения Name каждого товара, применив следующий синтаксис выборки:

```
static void ListProductNames(ProductInfo[] products)
{
  // Теперь получить только наименования товаров.
  Console.WriteLine("Only product names:");
  var names = from p in products select p.Name;

  foreach (var n in names)
  {
    Console.WriteLine("Name: {0}", n);
  }
}
```

Получение подмножества данных

Чтобы получить определенное подмножество из контейнера, можно использовать операцию where. Общий шаблон запроса становится таким:

```
var результат =
  from элемент in контейнер
  where булевскоеВыражение
  select элемент;
```

Обратите внимание, что операция where ожидает выражение, результатом вычисления которого является булевское значение. Например, чтобы извлечь из аргумента ProductInfo[] только товарные позиции, складские запасы которых составляют более 25 единиц, можно написать следующий код:

```
static void GetOverstock(ProductInfo[] products)
{
  Console.WriteLine("The overstock items!");

  // Получить только товары со складским запасом более 25 единиц.
  var overstock = from p in products where p.NumberInStock > 25 select p;

  foreach (ProductInfo c in overstock)
  {
    Console.WriteLine(c.ToString());
  }
}
```

Как демонстрировалось ранее в главе, при указании конструкции where разрешено применять любые операции C# для построения сложных выражений. Например, вспомните запрос, который извлекал только автомобили марки BMW, движущиеся со скоростью минимум 90 миль в час:

```
//Получить автомобили BMW, движущиеся со скоростью минимум 90 миль в час.
var onlyFastBMWs = from c in myCars
                   where c.Make == "BMW" && c.Speed >= 90 select c;
foreach (Car c in onlyFastBMWs)
{
  Console.WriteLine("{0} is going {1} MPH", c.PetName, c.Speed);
}
```

Проецирование в новые типы данных

Новые формы данных также можно проецировать из существующего источника данных. Давайте предположим, что необходимо принять входной параметр `ProductInfo[]` и получить результирующий набор, который учитывает только имя и описание каждого товара. Для этого понадобится определить оператор `select`, динамически выдающий новый анонимный тип:

```
static void GetNamesAndDescriptions(ProductInfo[] products)
{
  Console.WriteLine("Names and Descriptions:");
  var nameDesc = from p in products select new { p.Name, p.Description };

  foreach (var item in nameDesc)
  {
    // Можно было бы также использовать свойства Name
    // и Description напрямую.
    Console.WriteLine(item.ToString());
  }
}
```

Не забывайте, что когда запрос LINQ использует проекцию, нет никакого способа узнать лежащий в ее основе тип данных, т.к. он определяется на этапе компиляции. В подобных случаях ключевое слово `var` является обязательным. Кроме того, вспомните о невозможности создания методов с неявно типизированными возвращаемыми значениями. Таким образом, следующий метод не скомпилируется:

```
static var GetProjectedSubset(ProductInfo[] products)
{
  var nameDesc = from p in products select new { p.Name, p.Description };
  return nameDesc;  // Так поступать нельзя!
}
```

В случае необходимости возвращения спроецированных данных вызывающему коду один из подходов предусматривает трансформацию результата запроса в объект `System.Array` с применением расширяющего метода `ToArray()`. Следовательно, модифицировав выражение запроса, как показано ниже:

```
// Теперь возвращаемым значением является объект Array.
static Array GetProjectedSubset(ProductInfo[] products)
{
  var nameDesc = from p in products select new { p.Name, p.Description };

  // Отобразить набор анонимных объектов на объект Array.
  return nameDesc.ToArray();
}
```

метод `GetProjectedSubset()` можно вызвать и обработать возвращенные им данные:

```
Array objs = GetProjectedSubset(itemsInStock);
foreach (object o in objs)
{
  Console.WriteLine(o); // Вызывает метод ToString()
                        // на каждом анонимном объекте.
}
```

Как видите, здесь должен использоваться буквальный объект `System.Array`, а применять синтаксис объявления массива C# невозможно, учитывая, что лежащий в основе проекции тип неизвестен, поскольку речь идет об анонимном классе, который сгенерирован компилятором. Кроме того, параметр типа для обобщенного метода `ToArray<T>()` не указывается, потому что он тоже не известен вплоть до этапа компиляции.

Очевидная проблема связана с утратой строгой типизации, т.к. каждый элемент в объекте `Array` считается относящимся к типу `Object`. Тем не менее, когда нужно возвратить результирующий набор LINQ, который является результатом операции проецирования в анонимный тип, трансформация данных в тип `Array` (или другой подходящий контейнер через другие члены типа `Enumerable`) обязательна.

Проецирование в другие типы данных

В дополнение к проецированию в анонимные типы результаты запроса LINQ можно проецировать в другой конкретный тип, что позволяет применять статическую типизацию и реализацию `IEnumerable<T>` как результирующий набор. Для начала создайте уменьшенную версию класса `ProductInfo`:

```
namespace FunWithLinqExpressions
{
  class ProductInfoSmall
  {
    public string Name {get; set;} = "";
    public string Description {get; set;} = "";
    public override string ToString()
      => $"Name={Name}, Description={Description}";
  }
}
```

Следующее изменение касается проецирования результатов запроса в коллекцию объектов `ProductInfoSmall`, а не анонимных типов. Добавьте в класс `ProductInfoSmall` следующий метод:

```
static void GetNamesAndDescriptionsTyped(
    ProductInfo[] products)
{
  Console.WriteLine("Names and Descriptions:");
  IEnumerable<ProductInfoSmall> nameDesc =
    from p
    in products
    select new ProductInfoSmall
      { Name=p.Name, Description=p.Description };

  foreach (ProductInfoSmall item in nameDesc)
  {
    Console.WriteLine(item.ToString());
  }
}
```

При проецировании LINQ у вас есть выбор, какой метод использовать (в анонимные или в строго типизированные объекты). Решение, которое вы примете, полностью зависит от имеющихся бизнес-требований.

Подсчет количества с использованием класса Enumerable

Во время проецирования новых пакетов данных у вас может возникнуть необходимость выяснить количество элементов, возвращаемых внутри последовательности. Для определения числа элементов, которые возвращаются из выражения запроса LINQ, можно применять расширяющий метод `Count()` класса `Enumerable`. Например, следующий метод будет искать в локальном массиве все объекты `string`, которые имеют длину, превышающую шесть символов, и выводить их количество:

```
static void GetCountFromQuery()
{
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                "Fallout 3", "Daxter", "System Shock 2"};

  // Получить количество элементов из запроса.
  int numb =
    (from g in currentVideoGames where g.Length > 6 select g).Count();

  // Вывести количество элементов.
  Console.WriteLine("{0} items honor the LINQ query.", numb);
}
```

Изменение порядка следования элементов в результирующих наборах на противоположный

Изменить порядок следования элементов в результирующем наборе на противоположный довольно легко с помощью расширяющего метода `Reverse<T>()` класса `Enumerable`. Например, в показанном далее методе выбираются все элементы из входного параметра `ProductInfo[]` в обратном порядке:

```
static void ReverseEverything(ProductInfo[] products)
{
  Console.WriteLine("Product in reverse:");
  var allProducts = from p in products select p;
  foreach (var prod in allProducts.Reverse())
  {
    Console.WriteLine(prod.ToString());
  }
}
```

Выражения сортировки

В начальных примерах настоящей главы вы видели, что в выражении запроса может использоваться операция `orderby` для сортировки элементов в подмножестве по заданному значению. По умолчанию принят порядок по возрастанию, поэтому строки сортируются в алфавитном порядке, числовые значения — от меньшего к большему и т.д. Если вы хотите просматривать результаты в порядке по убыванию, просто включите в выражение запроса операцию `descending`. Взгляните на следующий метод:

```
static void AlphabetizeProductNames(ProductInfo[] products)
{
  // Получить названия товаров в алфавитном порядке.
  var subset = from p in products orderby p.Name select p;

  Console.WriteLine("Ordered by Name:");
```

```
  foreach (var p in subset)
  {
    Console.WriteLine(p.ToString());
  }
}
```

Хотя порядок по возрастанию является стандартным, свои намерения можно прояснить, явно указав операцию ascending:

```
var subset = from p in products orderby p.Name ascending select p;
```

Для получения элементов в порядке убывания служит операция descending:

```
var subset = from p in products orderby p.Name descending select p;
```

LINQ как лучшее средство построения диаграмм Венна

Класс Enumerable поддерживает набор расширяющих методов, которые позволяют применять два (или более) запроса LINQ в качестве основы для нахождения объединений, разностей, конкатенаций и пересечений данных. Первым мы рассмотрим расширяющий метод Except(). Он возвращает результирующий набор LINQ, содержащий разность между двумя контейнерами, которой в этом случае является значение Yugo:

```
static void DisplayDiff()
{
  List<string> myCars = new List<String> {"Yugo", "Aztec", "BMW"};
  List<string> yourCars = new List<String>{"BMW", "Saab", "Aztec" };
  var carDiff =
    (from c in myCars select c).Except(from c2 in yourCars select c2);
  Console.WriteLine("Here is what you don't have, but I do:");
  // Выводит Yugo.
  foreach (string s in carDiff)
  {
    Console.WriteLine(s);
  }
}
```

Метод Intersect() возвращает результирующий набор, который содержит общие элементы данных в наборе контейнеров. Например, следующий метод возвращает последовательность из Aztec и BMW:

```
static void DisplayIntersection()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
  // Получить общие члены.
  var carIntersect = (from c in myCars select c)
    .Intersect(from c2 in yourCars select c2);
  Console.WriteLine("Here is what we have in common:");
  // Выводит Aztec и BMW.
  foreach (string s in carIntersect)
  {
    Console.WriteLine(s);
  }
}
```

Метод `Union()` возвращает результирующий набор, который включает все члены множества запросов LINQ. Подобно любому объединению, даже если общий член встречается более одного раза, повторяющихся значений в результирующем наборе не будет. Следовательно, показанный ниже метод выведет на консоль значения `Yugo`, `Aztec`, `BMW` и `Saab`:

```
static void DisplayUnion()
{
  List<string> myCars = new List<string> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<string> { "BMW", "Saab", "Aztec" };

  // Получить объединение двух контейнеров.
  var carUnion =
      (from c in myCars select c).Union(from c2 in yourCars select c2);

  Console.WriteLine("Here is everything:");
  // Выводит все общие члены.
  foreach (string s in carUnion)
  {
    Console.WriteLine(s);
  }
}
```

Наконец, расширяющий метод `Concat()` возвращает результирующий набор, который является прямой конкатенацией результирующих наборов LINQ. Например, следующий метод выводит на консоль результаты `Yugo`, `Aztec`, `BMW`, `BMW`, `Saab` и `Aztec`:

```
static void DisplayConcat()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

  var carConcat = (from c in myCars select c)
    .Concat(from c2 in yourCars select c2);

  // Выводит Yugo Aztec BMW BMW Saab Aztec.
  foreach (string s in carConcat)
  {
    Console.WriteLine(s);
  }
}
```

Устранение дубликатов

При вызове расширяющего метода `Concat()` в результате очень легко получить избыточные элементы, и зачастую это может быть именно тем, что нужно. Однако в других случаях может понадобиться удалить дублированные элементы данных. Для этого необходимо просто вызвать расширяющий метод `Distinct()`:

```
static void DisplayConcatNoDups()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

  var carConcat =
     (from c in myCars select c).Concat(from c2 in yourCars select c2);
```

```
  // Выводит Yugo Aztec BMW Saab.
  foreach (string s in carConcat.Distinct())
  {
    Console.WriteLine(s);
  }
}
```

Операции агрегирования LINQ

Запросы LINQ могут также проектироваться для выполнения над результирующим набором разнообразных операций агрегирования. Одним из примеров может служить расширяющий метод `Count()`. Другие возможности включают получение среднего, максимального, минимального или суммы значений с использованием членов `Average()`, `Max()`, `Min()` либо `Sum()` класса `Enumerable`. Вот простой пример:

```
static void AggregateOps()
{
  double[] winterTemps = { 2.0, -21.3, 8, -4, 0, 8.2 };

  // Разнообразные примеры агрегации.
  // Выводит максимальную температуру:
  Console.WriteLine("Max temp: {0}",
                    (from t in winterTemps select t).Max());

  // Выводит минимальную температуру:
  Console.WriteLine("Min temp: {0}",
                    (from t in winterTemps select t).Min());

  // Выводит среднюю температуру:
  Console.WriteLine("Average temp: {0}",
                    (from t in winterTemps select t).Average());

  // Выводит сумму всех температур:
  Console.WriteLine("Sum of all temps: {0}",
                    (from t in winterTemps select t).Sum());
}
```

Приведенные примеры должны предоставить достаточный объем сведений, чтобы вы освоились с процессом построения выражений запросов LINQ. Хотя существуют дополнительные операции, которые пока еще не рассматривались, вы увидите примеры позже в книге, когда речь пойдет о связанных технологиях LINQ. В завершение вводного экскурса в LINQ оставшиеся материалы главы посвящены подробностям отношений между операциями запросов LINQ и лежащей в основе объектной моделью.

Внутреннее представление операторов запросов LINQ

К настоящему моменту вы уже знакомы с процессом построения выражений запросов с применением разнообразных операций запросов C# (таких как `from`, `in`, `where`, `orderby` и `select`). Вдобавок вы узнали, что определенная функциональность API-интерфейса LINQ to Objects доступна только через вызов расширяющих методов класса `Enumerable`. В действительности при компиляции запросов LINQ компилятор C# транслирует все операции LINQ в вызовы методов класса `Enumerable`.

Огромное количество методов класса Enumerable прототипированы для приема делегатов в качестве аргументов. Многие методы требуют обобщенный делегат по имени Func<>, который был описан во время рассмотрения обобщенных делегатов в главе 10. Взгляните на метод Where() класса Enumerable, вызываемый автоматически в случае использования операции where:

```
// Перегруженные версии метода Enumerable.Where<T>().
// Обратите внимание, что второй параметр имеет тип System.Func<>.
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source,
  System.Func<TSource,int,bool> predicate)

public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source,
  System.Func<TSource,bool> predicate)
```

Делегат Func<> представляет шаблон фиксированной функции с набором до 16 аргументов и возвращаемым значением. Если вы исследуете этот тип в браузере объектов Visual Studio, то заметите разнообразные формы делегата Func<>. Например:

```
// Различные формы делегата Func<>.
public delegate TResult Func<T1,T2,T3,T4,TResult>
    (T1 arg1, T2 arg2, T3 arg3, T4 arg4)

public delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)

public delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)

public delegate TResult Func<T1,TResult>(T1 arg1)

public delegate TResult Func<TResult>()
```

Учитывая, что многие члены класса System.Linq.Enumerable при вызове ожидают получить делегат, можно вручную создать новый тип делегата и написать для него необходимые целевые методы, применить анонимный метод C# или определить подходящее лямбда-выражение. Независимо от выбранного подхода конечный результат будет одним и тем же.

Хотя использование операций запросов LINQ является, несомненно, самым простым способом построения запросов LINQ, давайте взглянем на все возможные подходы, чтобы увидеть связь между операциями запросов C# и лежащим в основе типом Enumerable.

Построение выражений запросов с применением операций запросов

Для начала создадим новый проект консольного приложения по имени LinqUsingEnumerable. В классе Program будут определены статические вспомогательные методы (вызываемые внутри операторов верхнего уровня) для иллюстрации разнообразных подходов к построению выражений запросов LINQ.

Первый метод, QueryStringsWithOperators(), предлагает наиболее прямолинейный способ создания выражений запросов и идентичен коду примера LinqOverArray, который приводился ранее в главе:

```
using System.Linq;
static void QueryStringWithOperators()
{
  Console.WriteLine("***** Using Query Operators *****");
```

```
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  var subset = from game in currentVideoGames
               where game.Contains(" ") orderby game select game;

  foreach (string s in subset)
  {
    Console.WriteLine("Item: {0}", s);
  }
}
```

Очевидное преимущество использования операций запросов C# при построении выражений запросов заключается в том, что делегаты `Func<>` и вызовы методов `Enumerable` остаются вне поля зрения и внимания, т.к. выполнение необходимой трансляции возлагается на компилятор C#. Бесспорно, создание выражений LINQ с применением различных операций запросов (`from`, `in`, `where` или `orderby`) является наиболее распространенным и простым подходом.

Построение выражений запросов с использованием типа `Enumerable` и лямбда-выражений

Имейте в виду, что применяемые здесь операции запросов LINQ представляют собой сокращенные версии вызова расширяющих методов, определенных в типе `Enumerable`. Рассмотрим показанный ниже метод `QueryStringsWithEnumerable AndLambdas()`, который обрабатывает локальный массив строк, но на этот раз в нем напрямую используются расширяющие методы `Enumerable`:

```
static void QueryStringsWithEnumerableAndLambdas()
{
  Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");

  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // Построить выражение запроса с использованием расширяющих методов,
  // предоставленных типу Array через тип Enumerable.
  var subset = currentVideoGames
    .Where(game => game.Contains(" "))
    .OrderBy(game => game).Select(game => game);

  // Вывести результаты.
  foreach (var game in subset)
  {
    Console.WriteLine("Item: {0}", game);
  }
  Console.WriteLine();
}
```

Здесь сначала вызывается расширяющий метод `Where()` на строковом массиве `currentVideoGames`. Вспомните, что класс `Array` получает данный метод от класса `Enumerable`. Метод `Enumerable.Where()` требует параметра типа делегата `System.Func<T1, TResult>`. Первый параметр типа упомянутого делегата представляет совместимые с интерфейсом `IEnumerable<T>` данные для обработки (массив строк в рассматриваемом случае), а второй — результирующие данные метода, которые получаются от единственного оператора, вставленного в лямбда-выражение.

Возвращаемое значение метода `Where()` в приведенном примере кода скрыто от глаз, но "за кулисами" работа происходит с типом `OrderedEnumerable`. На объекте указанного типа вызывается обобщенный метод `OrderBy()`, который также принимает параметр типа делегата `Func<>`. Теперь производится передача всех элементов по очереди посредством подходящего лямбда-выражения. Результатом вызова `OrderBy()` является новая упорядоченная последовательность первоначальных данных.

И, наконец, осуществляется вызов метода `Select()` на последовательности, возвращенной `OrderBy()`, который в итоге дает финальный набор данных, сохраняемый в неявно типизированной переменной по имени `subset`.

Конечно, такой "длинный" запрос LINQ несколько сложнее для восприятия, чем предыдущий пример с операциями запросов LINQ. Без сомнения, часть сложности связана с объединением в цепочку вызовов посредством операции точки. Вот тот же самый запрос с выделением каждого шага в отдельный фрагмент (разбивать запрос на части можно разными способами):

```
static void QueryStringsWithEnumerableAndLambdas2()
{
  Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};
  // Разбить на части.
  var gamesWithSpaces = currentVideoGames.Where(game => game.Contains(" "));
  var orderedGames = gamesWithSpaces.OrderBy(game => game);
  var subset = orderedGames.Select(game => game);
  foreach (var game in subset)
  {
    Console.WriteLine("Item: {0}", game);
  }
  Console.WriteLine();
}
```

Как видите, построение выражения запроса LINQ с применением методов класса `Enumerable` напрямую приводит к намного более многословному запросу, чем в случае использования операций запросов C#. Кроме того, поскольку методы `Enumerable` требуют передачи делегатов в качестве параметров, обычно необходимо писать лямбда-выражения, чтобы обеспечить обработку входных данных внутренней целью делегата.

Построение выражений запросов с использованием типа `Enumerable` и анонимных методов

Учитывая, что лямбда-выражения C# — это просто сокращенный способ работы с анонимными методами, рассмотрим третье выражение запроса внутри вспомогательного метода `QueryStringsWithAnonymousMethods()`:

```
static void QueryStringsWithAnonymousMethods()
{
  Console.WriteLine("***** Using Anonymous Methods *****");
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};
```

```
  // Построить необходимые делегаты Func<>
  // с использованием анонимных методов.
  Func<string, bool> searchFilter =
     delegate(string game) { return game.Contains(" "); };
  Func<string, string> itemToProcess = delegate(string s) { return s; };

  // Передать делегаты в методы класса Enumerable.
  var subset = currentVideoGames.Where(searchFilter).OrderBy(itemToProcess)
                                .Select(itemToProcess);

  // Вывести результаты.
  foreach (var game in subset)
  {
    Console.WriteLine("Item: {0}", game);
  }
  Console.WriteLine();
}
```

Такой вариант выражения запроса оказывается еще более многословным из-за создания вручную делегатов `Func<>`, применяемых методами `Where()`, `OrderBy()` и `Select()` класса `Enumerable`. Положительная сторона данного подхода связана с тем, что синтаксис анонимных методов позволяет заключить всю обработку, выполняемую делегатами, в единственное определение метода. Тем не менее, этот метод функционально эквивалентен методам `QueryStringsWithEnumerableAndLambdas()` и `QueryStringsWithOperators()`, созданным в предшествующих разделах.

Построение выражений запросов с использованием типа `Enumerable` и низкоуровневых делегатов

Наконец, если вы хотите строить выражение запроса с применением *многословного подхода*, то можете отказаться от использования синтаксиса лямбда-выражений и анонимных методов и напрямую создавать цели делегатов для каждого типа `Func<>`. Ниже показана финальная версия выражения запроса, смоделированная внутри нового типа класса по имени `VeryComplexQueryExpression`:

```
class VeryComplexQueryExpression
{
  public static void QueryStringsWithRawDelegates()
  {
    Console.WriteLine("***** Using Raw Delegates *****");

    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
      "Fallout 3", "Daxter", "System Shock 2"};

    // Построить необходимые делегаты Func<>.
    Func<string, bool> searchFilter = new Func<string, bool>(Filter);
    Func<string, string> itemToProcess =
       new Func<string,string>(ProcessItem);

    // Передать делегаты в методы класса Enumerable.
    var subset =
      currentVideoGames
        .Where(searchFilter)
        .OrderBy(itemToProcess)
        .Select(itemToProcess);
```

```
    // Вывести результаты.
    foreach (var game in subset)
    {
      Console.WriteLine("Item: {0}", game);
    }
    Console.WriteLine();
  }
  // Цели делегатов.
  public static bool Filter(string game)
  {
    return game.Contains(" ");
  }

  public static string ProcessItem(string game)
  {
    return game;
  }
}
```

Чтобы протестировать такую версию логики обработки строк, метод `QueryStringsWithRawDelegates()` понадобится вызвать внутри операторов верхнего уровня в классе `Program`:

```
VeryComplexQueryExpression.QueryStringsWithRawDelegates();
```

Если теперь запустить приложение, чтобы опробовать все возможные подходы, вывод окажется идентичным независимо от выбранного пути. Запомните перечисленные ниже моменты относительно выражений запросов и их внутреннего представления.

- Выражения запросов создаются с применением разнообразных операций запросов C#.
- Операции запросов — это просто сокращенное обозначение для вызова расширяющих методов, определенных в типе `System.Linq.Enumerable`.
- Многие методы класса `Enumerable` требуют передачи делегатов (в частности, `Func<>`) в качестве параметров.
- Любой метод, ожидающий параметра типа делегата, может принимать вместо него лямбда-выражение.
- Лямбда-выражения являются всего лишь замаскированными анонимными методами (и значительно улучшают читабельность).
- Анонимные методы представляют собой сокращенные обозначения для размещения экземпляра низкоуровневого делегата и ручного построения целевого метода делегата.

Хотя здесь мы погрузились в детали чуть глубже, чем возможно хотелось, приведенное обсуждение должно было способствовать пониманию того, что фактически делают "за кулисами" дружественные к пользователю операции запросов C#.

Резюме

LINQ — это набор взаимосвязанных технологий, которые были разработаны для предоставления единого и симметричного стиля взаимодействия с данными несходных форм. Как объяснялось в главе, LINQ может взаимодействовать с любым типом, реализующим интерфейс `IEnumerable<T>`, в том числе с простыми массивами, а также с обобщенными и необобщенными коллекциями данных.

Было показано, что работа с технологиями LINQ обеспечивается несколькими средствами языка C#. Например, учитывая тот факт, что выражения запросов LINQ могут возвращать любое количество результирующих наборов, для представления лежащего в основе типа данных принято использовать ключевое слово `var`. Кроме того, для построения функциональных и компактных запросов LINQ могут применяться лямбда-выражения, синтаксис инициализации объектов и анонимные типы.

Более важно то, что операции запросов LINQ в C# на самом деле являются просто сокращенными обозначениями для обращения к статическим членам типа `System.Linq.Enumerable`. Вы узнали, что большинство членов класса `Enumerable` оперируют с типами делегатов `Func<T>` и для выполнения запроса могут принимать на входе адреса существующих методов, анонимные методы или лямбда-выражения.

ГЛАВА 14

Процессы, домены приложений и контексты загрузки

В настоящей главе будут представлены детали обслуживания сборки исполняющей средой, а также отношения между процессами, доменами приложений и контекстами загрузки.

Выражаясь кратко, *домены приложений* (Application Domain или просто AppDomain) представляют собой логические подразделы внутри заданного процесса, обслуживающего набор связанных сборок .NET Core. Как вы увидите, каждый домен приложения в дальнейшем подразделяется на *контекстные границы*, которые используются для группирования вместе похожих по смыслу объектов .NET Core. Благодаря понятию контекста исполняющая среда способна обеспечивать надлежащую обработку объектов со специальными требованиями.

Хотя вполне справедливо утверждать, что многие повседневные задачи программирования не предусматривают работу с процессами, доменами приложений или контекстами загрузки напрямую, их понимание важно при взаимодействии с многочисленными API-интерфейсами .NET Core, включая многопоточную и параллельную обработку, а также сериализацию объектов.

Роль процесса Windows

Концепция "процесса" существовала в операционных системах Windows задолго до выпуска платформы .NET/.NET Core. Пользуясь простыми терминами, *процесс* — это выполняющаяся программа. Тем не менее, формально процесс является концепцией уровня операционной системы, которая применяется для описания набора ресурсов (таких как внешние библиотеки кода и главный поток) и необходимых распределений памяти, используемой функционирующим приложением. Для каждого загруженного в память приложения .NET Core операционная система создает отдельный изолированный процесс для применения на протяжении всего времени его существования.

При использовании такого подхода к изоляции приложений в результате получается намного более надежная и устойчивая исполняющая среда, поскольку отказ одного процесса не влияет на работу других процессов. Более того, данные в одном процессе не доступны напрямую другим процессам, если только не применяются специфичные инструменты вроде пространства имен `System.IO.Pipes` или класса `MemoryMappedFile`.

Каждый процесс Windows получает уникальный идентификатор процесса (process identifier — PID) и может по мере необходимости независимо загружаться и выгружаться операционной системой (а также программно). Как вам возможно известно, в окне диспетчера задач Windows (открываемом по нажатию комбинации клавиш <Ctrl+Shift+Esc>) имеется вкладка Processes (Процессы), на которой можно просматривать разнообразные статические данные о процессах, функционирующих на машине. На вкладке Details (Подробности) можно видеть назначенный идентификатор PID и имя образа (рис. 14.1).

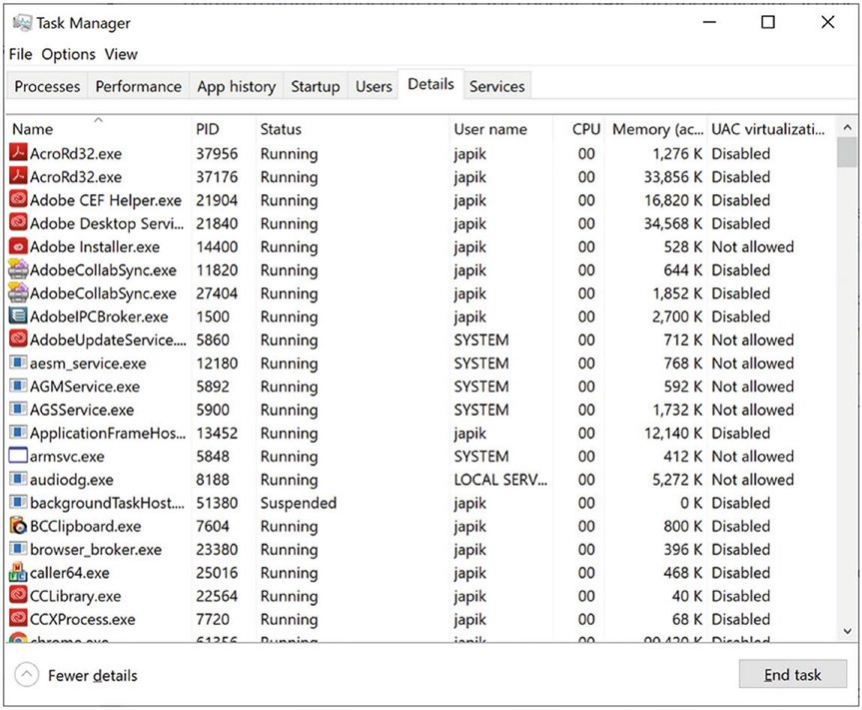

Рис. 14.1. Диспетчер задач Windows

Роль потоков

Каждый процесс Windows содержит начальный "поток", который действует как точка входа для приложения. Особенности построения многопоточных приложений на платформе .NET Core рассматриваются в главе 15; однако для понимания материала настоящей главы необходимо ознакомиться с несколькими рабочими определениями. *Поток* представляет собой путь выполнения внутри процесса. Выражаясь формально, первый поток, созданный точкой входа процесса, называется *главным потоком*. В любой программе .NET Core (консольном приложении, Windows-службе, приложении WPF и т.д.) точка входа помечается с помощью метода `Main()` или файла, содержащего операторы верхнего уровня. При обращении к этому коду автоматически создается главный поток.

Процессы, которые содержат единственный главный поток выполнения, по своей сути *безопасны в отношении потоков*, т.к. в каждый момент времени доступ к данным приложения может получать только один поток. Тем не менее, однопоточный процесс (особенно с графическим пользовательским интерфейсом) часто замедленно реагирует на действия пользователя, когда его единственный поток выполняет сложную операцию (наподобие печати длинного текстового файла, сложных математических вычислений или попытки подключения к удаленному серверу, находящемуся на расстоянии тысяч километров).

Учитывая такой потенциальный недостаток однопоточных приложений, операционные системы, которые поддерживаются .NET Core, и сама платформа .NET Core предоставляют главному потоку возможность порождения дополнительных вторичных потоков (называемых *рабочими потоками*) с использованием нескольких функций из API-интерфейса Windows, таких как `CreateThread()`. Каждый поток (первичный или вторичный) становится уникальным путем выполнения в процессе и имеет параллельный доступ ко всем совместно используемым элементам данных внутри этого процесса.

Нетрудно догадаться, что разработчики обычно создают дополнительные потоки для улучшения общей степени отзывчивости программы. Многопоточные процессы обеспечивают иллюзию того, что выполнение многочисленных действий происходит более или менее одновременно. Например, приложение может порождать дополнительный рабочий поток для выполнения трудоемкой единицы работы (вроде вывода на печать крупного текстового файла). После запуска вторичного потока главный поток продолжает реагировать на пользовательский ввод, что дает всему процессу возможность достигать более высокой производительности. Однако на самом деле так происходит не всегда: применение слишком большого количества потоков в одном процессе может приводить к *ухудшению* производительности из-за того, что центральный процессор должен переключаться между активными потоками внутри процесса (а это отнимает время).

На некоторых машинах многопоточность по большей части является иллюзией, обеспечиваемой операционной системой. Машины с единственным (не поддерживающим гиперпотоки) центральным процессором не обладают возможностью обработки множества потоков в одно и то же время. Взамен один центральный процессор выполняет по одному потоку за единицу времени (называемую *квантом времени*), частично основываясь на приоритете потока. По истечении выделенного кванта времени выполнение существующего потока приостанавливается, позволяя выполнять работу другому потоку. Чтобы поток не "забывал", что происходило до того, как его выполнение было приостановлено, ему предоставляется возможность записывать данные в локальное хранилище потоков (Thread Local Storage — TLS) и выделяется отдельный стек вызовов (рис. 14.2).

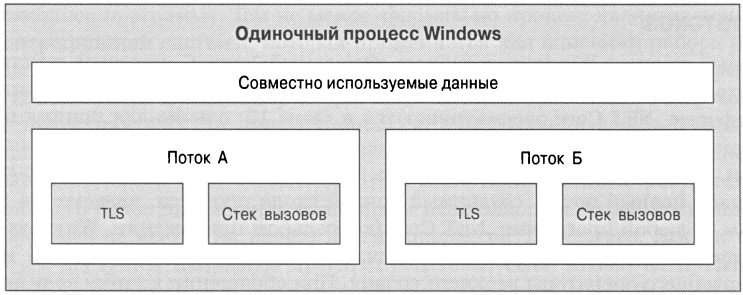

Рис. 14.2. Отношения между потоками и процессами в Windows

Если тема потоков для вас нова, то не стоит беспокоиться о деталях. На данном этапе просто запомните, что любой поток представляет собой уникальный путь выполнения внутри процесса Windows. Каждый процесс имеет главный поток (созданный посредством точки входа исполняемого файла) и может содержать дополнительные потоки, которые создаются программно.

Взаимодействие с процессами, используя платформу .NET Core

Несмотря на то что с процессами и потоками не связано ничего нового, способ взаимодействия с ними в рамках платформы .NET Core значительно изменился (в лучшую сторону). Чтобы подготовить почву для понимания области построения многопоточных сборок (см. главу 15), давайте начнем с выяснения способов взаимодействия с процессами, используя библиотеки базовых классов .NET Core.

В пространстве имен `System.Diagnostics` определено несколько типов, которые позволяют программно взаимодействовать с процессами и разнообразными типами, связанными с диагностикой, такими как журнал событий системы и счетчики производительности. В текущей главе нас интересуют только типы, связанные с процессами, которые описаны в табл. 14.1.

Таблица 14.1. Избранные типы пространства имен `System.Diagnostics`

Тип, связанный с процессами	Описание
`Process`	Предоставляет доступ к локальным и удаленным процессам, а также позволяет программно запускать и останавливать процессы
`ProcessModule`	Представляет модуль (`*.dll` или `*.exe`), загруженный в определенный процесс. Важно понимать, что тип `ProcessModule` может представлять *любой* модуль, т.е. двоичные сборки, основанные на COM, .NET или традиционном языке C
`ProcessModuleCollection`	Предоставляет строго типизированную коллекцию объектов `ProcessModule`
`ProcessStartInfo`	Указывает набор значений, применяемых при запуске процесса с помощью метода `Process.Start()`
`ProcessThread`	Представляет поток внутри заданного процесса. Имейте в виду, что тип `ProcessThread` используется для диагностирования набора потоков процесса, но не для порождения новых потоков выполнения в рамках процесса
`ProcessThreadCollection`	Предоставляет строго типизованную коллекцию объектов `ProcessThread`

Класс `System.Diagnostics.Process` позволяет анализировать процессы, выполняющиеся на заданной машине (локальные или удаленные). В классе `Process` также определены члены, предназначенные для программного запуска и завершения процессов, просмотра (или модификации) уровня приоритета процесса и получения списка активных потоков и/или загруженных модулей внутри указанного процесса. В табл. 14.2 перечислены некоторые основные свойства класса `System.Diagnostics.Process`.

Таблица 14.2. Избранные свойства класса `Process`

Свойство	Описание
`ExitTime`	Позволяет извлекать отметку времени, ассоциированную с процессом, который был завершен (представленную с помощью типа `DateTime`)
`Handle`	Возвращает дескриптор (представленный типом `IntPtr`), который был назначен процессу операционной системой. Это может быть полезно при построении приложений .NET, нуждающихся во взаимодействии с неуправляемым кодом
`Id`	Позволяет получать идентификатор PID связанного процесса
`MachineName`	Позволяет получать имя компьютера, на котором выполняется связанный процесс
`MainWindowTitle`	Позволяет получать заголовок главного окна процесса (если у процесса нет главного окна, то возвращается пустая строка)
`Modules`	Предоставляет доступ к строго типизованной коллекции `ProcessModuleCollection`, представляющей набор модулей (`*.dll` или `*.exe`), которые были загружены внутри текущего процесса
`ProcessName`	Позволяет получать имя процесса (которое, как и можно было предполагать, представляет собой имя самого приложения)
`Responding`	Позволяет получать значение, которое указывает, реагирует ли пользовательский интерфейс процесса на пользовательский ввод (или в текущий момент находится в "зависшем" состоянии)
`StartTime`	Позволяет получать значение времени, когда был запущен процесс (представленное с помощью типа `DateTime`)
`Threads`	Позволяет получать набор потоков, выполняемых в связанном процессе (представленный посредством коллекции объектов `ProcessThread`)

Кроме перечисленных выше свойств в классе `System.Diagnostics.Process` определено несколько полезных методов (табл. 14.3).

Таблица 14.3. Избранные методы класса `Process`

Метод	Описание
`CloseMainWindow()`	Этот метод закрывает процесс, который содержит пользовательский интерфейс, отправляя его главному окну сообщение о закрытии
`GetCurrentProcess()`	Этот статический метод возвращает новый объект `Process`, который представляет процесс, активный в текущий момент
`GetProcesses()`	Этот статический метод возвращает массив объектов `Process`, представляющих процессы, которые выполняются на заданной машине
`Kill()`	Этот метод немедленно останавливает связанный процесс
`Start()`	Этот метод запускает процесс

Перечисление выполняющихся процессов

Для иллюстрации способа манипулирования объектами `Process` создайте новый проект консольного приложения C# по имени `ProcessManipulator` и определите в классе `Program` следующий вспомогательный статический метод (не забудьте импортировать в файл кода пространства имен `System.Diagnostics` и `System.Linq`):

```
static void ListAllRunningProcesses()
{
  // Получить все процессы на локальной машине, упорядоченные по PID.
  var runningProcs = from proc in Process.GetProcesses(".")
                     orderby proc.Id select proc;

  // Вывести для каждого процесса идентификатор PID и имя.
  foreach(var p in runningProcs)
  {
    string info = $"-> PID: {p.Id}\tName: {p.ProcessName}";
    Console.WriteLine(info);
  }
  Console.WriteLine("*************************************\n");
}
```

Статический метод `Process.GetProcesses()` возвращает массив объектов `Process`, которые представляют выполняющиеся процессы на целевой машине (передаваемая методу строка `"."` обозначает локальный компьютер). После получения массива объектов `Process` можно обращаться к любым членам, описанным в табл. 14.2 и 14.3. Здесь просто для каждого процесса выводятся идентификатор PID и имя с упорядочением по PID. Модифицируйте операторы верхнего уровня, как показано ниже:

```
using System;
using System.Diagnostics;
using System.Linq;

Console.WriteLine("***** Fun with Processes *****\n");
ListAllRunningProcesses();
Console.ReadLine();
```

Запустив приложение, вы увидите список имен и идентификаторов PID для всех процессов на локальной машине. Ниже показана часть вывода (ваш вывод наверняка будет отличаться):

```
***** Fun with Processes *****

-> PID: 0 Name: Idle
-> PID: 4 Name: System
-> PID: 104 Name: Secure System
-> PID: 176 Name: Registry
-> PID: 908 Name: svchost
-> PID: 920 Name: smss
-> PID: 1016 Name: csrss
-> PID: 1020 Name: NVDisplay.Container
-> PID: 1104 Name: wininit
-> PID: 1112 Name: csrss
*************************************
```

Исследование конкретного процесса

В дополнение к полному списку всех выполняющихся процессов на заданной машине статический метод `Process.GetProcessById()` позволяет получать одиночный объект `Process` по ассоциированному с ним идентификатору PID. В случае запроса несуществующего PID генерируется исключение `ArgumentException`. Например, чтобы получить объект `Process`, который представляет процесс с PID, равным 30592, можно написать следующий код:

```
// Если процесс с PID, равным 30592, не существует,
// то сгенерируется исключение во время выполнения.
static void GetSpecificProcess()
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(30592);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

К настоящему моменту вы уже знаете, как получить список всех процессов, а также специфический процесс на машине посредством поиска по PID. Наряду с выяснением идентификаторов PID и имен процессов класс `Process` позволяет просматривать набор текущих потоков и библиотек, применяемых внутри заданного процесса. Давайте посмотрим, как это делается.

Исследование набора потоков процесса

Набор потоков представлен в виде строго типизованной коллекции `ProcessThreadCollection`, которая содержит определенное количество отдельных объектов `ProcessThread`. В целях иллюстрации добавьте к текущему приложению приведенный далее вспомогательный статический метод:

```
static void EnumThreadsForPid(int pID)
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(pID);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }
  // Вывести статистические сведения по каждому потоку в указанном процессе
  Console.WriteLine("Here are the threads used by: {0}",
    theProc.ProcessName);
  ProcessThreadCollection theThreads = theProc.Threads;
```

```
  foreach(ProcessThread pt in theThreads)
  {
    string info =
     $"-> Thread ID: {pt.Id}\tStart Time: {pt.StartTime.ToShortTimeString()}\
tPriority:{pt.PriorityLevel}";
    Console.WriteLine(info);
  }
  Console.WriteLine("**********************************\n");
}
```

Как видите, свойство `Threads` в типе `System.Diagnostics.Process` обеспечивает доступ к классу `ProcessThreadCollection`. Здесь для каждого потока внутри указанного клиентом процесса выводится назначенный идентификатор потока, время запуска и уровень приоритета. Обновите операторы верхнего уровня в своей программе, чтобы запрашивать у пользователя идентификатор PID процесса, подлежащего исследованию:

```
static void Main(string[] args)
{
  ...
  // Запросить у пользователя PID и вывести набор активных потоков.
  Console.WriteLine("***** Enter PID of process to investigate *****");
  Console.Write("PID: ");
  string pID = Console.ReadLine();
  int theProcID = int.Parse(pID);

  EnumThreadsForPid(theProcID);
  Console.ReadLine();
}
```

После запуска приложения можно вводить PID любого процесса на машине и просматривать имеющиеся внутри него потоки. В следующем выводе показан неполный список потоков, используемых процессом с PID 3804, который (так случилось) обслуживает браузер Edge:

```
***** Enter PID of process to investigate *****
PID: 3804
Here are the threads used by: msedge
-> Thread ID: 3464 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 19420 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 17780 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 22380 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 27580 Start Time: 01:20 PM Priority: -4
...
**********************************
```

Помимо `Id`, `StartTime` и `PriorityLevel` тип `ProcessThread` содержит дополнительные члены, наиболее интересные из которых перечислены в табл. 14.4.

Прежде чем двигаться дальше, необходимо уяснить, что тип `ProcessThread` *не является* сущностью, применяемой для создания, приостановки или уничтожения потоков на платформе .NET Core. Тип `ProcessThread` скорее представляет собой средство, позволяющее получать диагностическую информацию по активным потокам Windows внутри выполняющегося процесса. Более подробные сведения о том, как создавать многопоточные приложения с использованием пространства имен `System.Threading`, приводятся в главе 15.

Таблица 14.4. Избранные члены типа `ProcessThread`

Член	Описание
`CurrentPriority`	Получает текущий приоритет потока
`Id`	Получает уникальный идентификатор потока
`IdealProcessor`	Устанавливает предпочитаемый процессор для выполнения заданного потока
`PriorityLevel`	Получает или устанавливает уровень приоритета потока
`ProcessorAffinity`	Устанавливает процессоры, на которых может выполняться связанный поток
`StartAddress`	Получает адрес в памяти функции, вызванной операционной системой, которая запустила данный поток
`StartTime`	Получает время, когда операционная система запустила поток
`ThreadState`	Получает текущее состояние потока
`TotalProcessorTime`	Получает общее время использования процессора данным потоком
`WaitReason`	Получает причину, по которой поток находится в состоянии ожидания

Исследование набора модулей процесса

Теперь давайте посмотрим, как реализовать проход по загруженным модулям, которые размещены внутри конкретного процесса. Когда речь идет о процессах, *модуль* —это общий термин, применяемый для описания заданной сборки `*.dll` (или самого файла `*.exe`), которая обслуживается специфичным процессом. Когда производится доступ к коллекции `ProcessModuleCollection` через свойство `Process.Modules`, появляется возможность перечисления *всех модулей*, размещенных внутри процесса: библиотек на основе .NET Core, COM и традиционного языка C. Взгляните на показанный ниже дополнительный вспомогательный метод, который будет перечислять модули в процессе с указанным идентификатором PID:

```
static void EnumModsForPid(int pID)
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(pID);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }
  Console.WriteLine("Here are the loaded modules for: {0}",
                    theProc.ProcessName);
  ProcessModuleCollection theMods = theProc.Modules;
```

```
    foreach(ProcessModule pm in theMods)
    {
      string info = $"-> Mod Name: {pm.ModuleName}";
      Console.WriteLine(info);
    }
    Console.WriteLine("*********************************\n");
  }
```

Чтобы получить какой-то вывод, давайте просмотрим загружаемые модули для процесса, обслуживающего программу текущего примера (`ProcessManipulator`). Для этого нужно запустить приложение, выяснить идентификатор PID, назначенный `ProcessManipulator.exe` (посредством диспетчера задач), и передать значение PID методу `EnumModsForPid()`. Вас может удивить, что с простым консольным приложением связан настолько внушительный список библиотек `*.dll` (`GDI32.dll`, `USER32.dll`, `ole32.dll` и т.д.). Ниже показан частичный список загруженных модулей (ради краткости отредактированный):

```
Here are (some of) the loaded modules for: ProcessManipulator
Here are the loaded modules for: ProcessManipulator
-> Mod Name: ProcessManipulator.exe
-> Mod Name: ntdll.dll
-> Mod Name: KERNEL32.DLL
-> Mod Name: KERNELBASE.dll
-> Mod Name: USER32.dll
-> Mod Name: win32u.dll
-> Mod Name: GDI32.dll
-> Mod Name: gdi32full.dll
-> Mod Name: msvcp_win.dll
-> Mod Name: ucrtbase.dll
-> Mod Name: SHELL32.dll
-> Mod Name: ADVAPI32.dll
-> Mod Name: msvcrt.dll
-> Mod Name: sechost.dll
-> Mod Name: RPCRT4.dll
-> Mod Name: IMM32.DLL
-> Mod Name: hostfxr.dll
-> Mod Name: hostpolicy.dll
-> Mod Name: coreclr.dll
-> Mod Name: ole32.dll
-> Mod Name: combase.dll
-> Mod Name: OLEAUT32.dll
-> Mod Name: bcryptPrimitives.dll
-> Mod Name: System.Private.CoreLib.dll
...
*********************************
```

Запуск и останов процессов программным образом

Финальными аспектами класса `System.Diagnostics.Process`, которые мы здесь исследуем, являются методы `Start()` и `Kill()`. Они позволяют программно запускать и завершать процесс. В качестве примера создадим вспомогательный статический метод `StartAndKillProcess()` с приведенным ниже кодом.

На заметку! В зависимости от настроек операционной системы, касающихся безопасности, для запуска новых процессов могут требоваться права администратора.

```
static void StartAndKillProcess()
{
  Process proc = null;
  // Запустить Edge и перейти на Facebook!
  try
  {
    proc = Process.Start(@"C:\Program Files (x86)\Microsoft\Edge\
Application\msedge.exe",
      "www.facebook.com");
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine(ex.Message);
  }
  // Уничтожить процесс по нажатию <Enter>.
  Console.Write("--> Hit enter to kill {0}...",
    proc.ProcessName);
  Console.ReadLine();
  // Уничтожить все процессы msedge.exe.
  try
  {
    foreach (var p in Process.GetProcessesByName("MsEdge"))
    {
      p.Kill(true);
    }
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Статический метод `Process.Start()` имеет несколько перегруженных версий. Как минимум, понадобится указать путь и имя файла запускаемого процесса. В рассматриваемом примере используется версия метода `Start()`, которая позволяет задавать любые дополнительные аргументы, подлежащие передаче в точку входа программы, в данном случае веб-страницу для загрузки.

В результате вызова метода `Start()` возвращается ссылка на новый активизированный процесс. Чтобы завершить данный процесс, потребуется просто вызвать метод `Kill()` уровня экземпляра. Поскольку Microsoft Edge запускает множество процессов, для их уничтожения организован цикл. Вызовы `Start()` и `Kill()` помещены внутрь блока `try/catch` с целью обработки исключений `InvalidOperationException`. Это особенно важно при вызове метода `Kill()`, потому что такое исключение генерируется, если процесс был завершен до вызова `Kill()`.

На заметку! В .NET Framework (до выхода .NET Core) для запуска процесса методу `Process.Start()` можно было передавать либо полный путь и имя файла процесса, либо его ярлык операционной системы (например, `msedge`). С появлением .NET Core и межплатформенной поддержки должны указываться полный путь и имя файла процесса. Файловые ассоциации операционной системы можно задействовать с применением класса `ProcessStartInfo`, раскрываемого в последующих двух разделах.

Управление запуском процесса с использованием класса `ProcessStartInfo`

Метод `Process.Start()` позволяет также передавать объект типа `System.Diagnostics.ProcessStartInfo` для указания дополнительной информации, касающейся запуска определенного процесса. Ниже приведено частичное определение `ProcessStartInfo` (полное определение можно найти в документации):

```
public sealed class ProcessStartInfo : object
{
  public ProcessStartInfo();
  public ProcessStartInfo(string fileName);
  public ProcessStartInfo(string fileName, string arguments);
  public string Arguments { get; set; }
  public bool CreateNoWindow { get; set; }
  public StringDictionary EnvironmentVariables { get; }
  public bool ErrorDialog { get; set; }
  public IntPtr ErrorDialogParentHandle { get; set; }
  public string FileName { get; set; }
  public bool LoadUserProfile { get; set; }
  public SecureString Password { get; set; }
  public bool RedirectStandardError { get; set; }
  public bool RedirectStandardInput { get; set; }
  public bool RedirectStandardOutput { get; set; }
  public Encoding StandardErrorEncoding { get; set; }
  public Encoding StandardOutputEncoding { get; set; }
  public bool UseShellExecute { get; set; }
  public string Verb { get; set; }
  public string[] Verbs { get; }
  public ProcessWindowStyle WindowStyle { get; set; }
  public string WorkingDirectory { get; set; }
}
```

Чтобы опробовать настройку запуска процесса, модифицируйте метод `StartAndKillProcess()` для загрузки Microsoft Edge и перехода на сайт `www.facebook.com` с применением ассоциации `MsEdge`:

```
static void StartAndKillProcess()
{
  Process proc = null;
  // Запустить Microsoft Edge и перейти на сайт Facebook
  // с развернутым на весь экран окном.
  try
  {
    ProcessStartInfo startInfo = new
      ProcessStartInfo("MsEdge", "www.facebook.com");
```

```
    startInfo.UseShellExecute = true;
    proc = Process.Start(startInfo);
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine(ex.Message);
  }
  ...
}
```

В .NET Core свойство `UseShellExecute` по умолчанию имеет значение `false`, тогда как в предшествующих версиях .NET его стандартным значением было `true`. Именно по этой причине показанная ниже предыдущая версия `Process.Start()` больше не работает без использования `ProcessStartInfo` и установки свойства `UseShellExecute` в `true`:

```
Process.Start("msedge")
```

Использование команд операционной системы с классом `ProcessStartInfo`

Помимо применения ярлыков операционной системы для запуска приложений с классом `ProcessStartInfo` можно также использовать файловые ассоциации. Если в среде Windows щелкнуть правой кнопкой мыши на имени документа Word, то с помощью контекстного меню можно будет отредактировать или распечатать этот документ. Давайте посредством класса `ProcessStartInfo` выясним доступные команды и затем применим их для манипулирования процессом. Создайте новый метод со следующим кодом:

```
static void UseApplicationVerbs()
{
  int i = 0;
  // Укажите здесь фактический путь и имя документа на своей машине.
  ProcessStartInfo si =
    new ProcessStartInfo(@"..\TestPage.docx");
  foreach (var verb in si.Verbs)
  {
    Console.WriteLine($" {i++}. {verb}");
  }
  si.WindowStyle = ProcessWindowStyle.Maximized;
  si.Verb = "Edit";
  si.UseShellExecute = true;
  Process.Start(si);
}
```

Первая часть кода выводит все команды, доступные для документа Word:

```
***** Fun with Processes *****
0. Edit
1. OnenotePrintto
2. Open
3. OpenAsReadOnly
4. Print
5. Printto
6. ViewProtected
```

После установки `WindowStyle` в `Maximized` (т.е. развернутое на весь экран окно) команда (`Verb`) устанавливается в `Edit`, что приводит к открытию документа в режиме редактирования. В случае установки команды в `Print` документ будет отправлен прямо на принтер.

Теперь, когда вы понимаете роль процессов Windows и знаете способы взаимодействия с ними из кода C#, можно переходить к исследованию концепции доменов приложений .NET.

На заметку! Каталог, в котором выполняется приложение, зависит от того, как вы его запускаете. Если вы применяете команду `dotnet run`, то текущим каталогом будет тот, где располагается файл проекта. Если же вы используете Visual Studio, тогда текущим будет каталог, в котором находится скомпилированная сборка, т.е. `.\bin\debug\net5.0`. Вам необходимо должным образом скорректировать путь к документу Word.

Домены приложений .NET

На платформах .NET и .NET Core исполняемые файлы не размещаются прямо внутри процесса Windows, как в случае традиционных неуправляемых приложений. Взамен исполняемый файл .NET и .NET Core попадает в отдельный логический раздел внутри процесса, который называется *доменом приложения*. Такое дополнительное разделение традиционного процесса Windows обеспечивает несколько преимуществ.

- Домены приложений являются ключевым аспектом нейтральной к операционным системам природы платформы .NET Core, поскольку такое логическое разделение абстрагирует отличия в том, как лежащая в основе операционная система представляет загруженный исполняемый файл.
- Домены приложений оказываются гораздо менее затратными в смысле вычислительных ресурсов и памяти по сравнению с полноценными процессами. Таким образом, среда CoreCLR способна загружать и выгружать домены приложений намного быстрее, чем формальный процесс, и может значительно улучшить масштабируемость серверных приложений.

Отдельный домен приложения полностью изолирован от других доменов приложений внутри процесса. Учитывая такой факт, имейте в виду, что приложение, выполняющееся в одном домене приложения, не может получать данные любого рода (глобальные переменные или статические поля) из другого домена приложения, если только не применяется какой-нибудь протокол распределенного программирования.

На заметку! Поддержка доменов приложений в .NET Core изменилась. В среде .NET Core существует в точности один домен приложения. Создавать новые домены приложений больше нельзя, поскольку это требует поддержки со стороны исполняющей среды и в общем случае сопряжено с высокими накладными расходами. Изоляцию сборок в .NET Core обеспечивает класс `ApplicationLoadContext` (рассматриваемый далее в главе).

Класс `System.AppDomain`

С выходом версии .NET Core класс `AppDomain` считается почти полностью устаревшим. Хотя большая часть оставшейся поддержки предназначена для упрощения перехода из .NET 4.x в .NET Core, она по-прежнему может приносить пользу, как объясняется в последующих двух разделах.

Взаимодействие со стандартным доменом приложения

С помощью статического свойства AppDomain.CurrentDomain можно получать доступ к стандартному домену приложения. При наличии такой точки доступа появляется возможность использования методов и свойств AppDomain для проведения диагностики во время выполнения.

Чтобы научиться взаимодействовать со стандартным доменом приложения, начните с создания нового проекта консольного приложения по имени DefaultAppDomainApp. Модифицируйте файл Program.cs, поместив в него следующий код, который просто выводит подробные сведения о стандартном домене приложения с применением нескольких членов класса AppDomain:

```
using System;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Runtime.Loader;

Console.WriteLine("***** Fun with the default AppDomain *****\n");
DisplayDADStats();
Console.ReadLine();

static void DisplayDADStats()
{
  // Получить доступ к домену приложения для текущего потока.
  AppDomain defaultAD = AppDomain.CurrentDomain;

  // Вывести разнообразные статистические данные об этом домене.
  Console.WriteLine("Name of this domain: {0}", defaultAD.FriendlyName);
                  // Дружественное имя этого домена
  Console.WriteLine("ID of domain in this process: {0}", defaultAD.Id);
                  // Идентификатор этого процесса
  Console.WriteLine("Is this the default domain?: {0}",
                      defaultAD.IsDefaultAppDomain());
                  // Является ли этот домен стандартным
  Console.WriteLine("Base directory of this domain: {0}",
                      defaultAD.BaseDirectory);
                  // Базовый каталог этого домена
  Console.WriteLine("Setup Information for this domain:");
                  // Информация о настройке этого домена
  Console.WriteLine("\tApplication Base: {0}",
                     defaultAD.SetupInformation.ApplicationBase);
                  // Базовый каталог приложения
  Console.WriteLine("\tTarget Framework: {0}",
                      defaultAD.SetupInformation.TargetFrameworkName);
                  // Целевая платформа
}
```

Ниже приведен вывод:

```
***** Fun with the default AppDomain *****

Name of this domain: DefaultAppDomainApp
ID of domain in this process: 1
Is this the default domain?: True
Base directory of this domain: C:\GitHub\Books\csharp8-wf\Code\Chapter_14\
```

```
DefaultAppDomainApp\DefaultAppDomainApp\bin\Debug\net5.0\
Setup Information for this domain:
  Application Base: C:\GitHub\Books\csharp8-wf\Code\Chapter_14
\DefaultAppDomainApp\DefaultAppDomainApp\bin\Debug\net5.0\
  Target Framework: .NETCoreApp,Version=v5.0
```

Обратите внимание, что имя стандартного домена приложения будет идентичным имени содержащегося внутри него исполняемого файла (`DefaultAppDomainApp.exe` в этом примере). Кроме того, значение базового каталога, которое будет использоваться для зондирования обязательных внешних закрытых сборок, отображается на текущее местоположение развернутого исполняемого файла.

Перечисление загруженных сборок

С применением метода `GetAssemblies()` уровня экземпляра можно просмотреть все сборки .NET Core, загруженные в указанный домен приложения. Метод возвращает массив объектов типа `Assembly` (рассматриваемого в главе 17). Для этого вы должны импортировать пространство имен `System.Reflection` в свой файл кода (как делали ранее).

В целях иллюстрации определите в классе `Program` новый вспомогательный метод по имени `ListAllAssembliesInAppDomain()`. Он будет получать список всех загруженных сборок и выводить для каждой из них дружественное имя и номер версии:

```
static void ListAllAssembliesInAppDomain()
{
  // Получить доступ к домену приложения для текущего потока.
  AppDomain defaultAD = AppDomain.CurrentDomain;
  // Извлечь все сборки, загруженные в стандартный домен приложения.
  Assembly[] loadedAssemblies = defaultAD.GetAssemblies();
  Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
                    defaultAD.FriendlyName);
  foreach(Assembly a in loadedAssemblies)
  {
    // Вывести имя и версию.
    Console.WriteLine($"-> Name,
                  Version: {a.GetName().Name}:{a.GetName().Version}");
  }
}
```

Добавив к операторам верхнего уровня вызов метода `ListAllAssembliesIn AppDomain()`, вы увидите, что в домене приложения, обслуживающем вашу исполняемую сборку, используются следующие библиотеки .NET Core:

```
***** Here are the assemblies loaded in DefaultAppDomainApp *****

-> Name, Version: System.Private.CoreLib:5.0.0.0
-> Name, Version: DefaultAppDomainApp:1.0.0.0
-> Name, Version: System.Runtime:5.0.0.0
-> Name, Version: System.Console:5.0.0.0
-> Name, Version: System.Threading:5.0.0.0
-> Name, Version: System.Text.Encoding.Extensions:5.0
```

Важно понимать, что список загруженных сборок может изменяться в любой момент по мере написания нового кода C#. Например, предположим, что метод `ListAllAssembliesInAppDomain()` модифицирован так, чтобы задействовать запрос LINQ, который упорядочивает загруженные сборки по имени:

```
using System.Linq;
static void ListAllAssembliesInAppDomain()
{
  // Получить доступ к домену приложения для текущего потока.
  AppDomain defaultAD = AppDomain.CurrentDomain;

  // Извлечь все сборки, загруженные в стандартный домен приложения.
  var loadedAssemblies =
    defaultAD.GetAssemblies().OrderBy(x=>x.GetName().Name);
  Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
    defaultAD.FriendlyName);
  foreach(Assembly a in loadedAssemblies)
  {
    // Вывести имя и версию.
    Console.WriteLine($"-> Name, Version: {a.GetName().Name}:{a.
GetName().Version}");
  }
}
```

Запустив приложение еще раз, вы заметите, что в память также была загружена сборка `System.Linq.dll`:

```
***** Here are the assemblies loaded in DefaultAppDomainApp *****

-> Name, Version: DefaultAppDomainApp:1.0.0.0
-> Name, Version: System.Console:5.0.0.0
-> Name, Version: System.Linq:5.0.0.0
-> Name, Version: System.Private.CoreLib:5.0.0.0
-> Name, Version: System.Runtime:5.0.0.0
-> Name, Version: System.Text.Encoding.Extensions:5.0.0.0
-> Name, Version: System.Threading:5.0.0
```

Изоляция сборок с помощью контекстов загрузки приложений

Как вам уже известно, домены приложений представляют собой логические разделы, используемые для обслуживания сборок .NET Core. Кроме того, домен приложения может быть дополнительно разделен на многочисленные границы контекстов загрузки. Концептуально контекст загрузки создает область видимости для загрузки, распознавания и потенциально выгрузки набора сборок. По существу контекст загрузки .NET Core наделяет одиночный домен приложения возможностью установить "конкретный дом" для заданного объекта.

На заметку! Хотя понимать процессы и домены приложений довольно-таки важно, в большинстве приложений .NET Core никогда не потребуется работать с контекстами загрузки. Этот обзорный материал был включен в книгу только ради того, чтобы представить более полную картину.

Класс `AssemblyLoadContext` позволяет загружать дополнительные сборки в их собственные контексты. В целях демонстрации создайте новый проект библиотеки классов по имени `ClassLibary1` и добавьте его к текущему решению. С использованием интерфейса командной строки .NET Core CLI выполните показанные ниже команды в каталоге, содержащем текущее решение:

```
dotnet new classlib -lang c# -n ClassLibrary1 -o .\ClassLibrary1 -f net5.0
dotnet sln .\Chapter14_AllProjects.sln add .\ClassLibrary1
```

Затем добавьте в `DefaultAppDomainApp` ссылку на проект `ClassLibrary1`, выполнив следующую команду CLI:

```
dotnet add DefaultAppDomainApp reference ClassLibrary1
```

Если вы работаете в Visual Studio, тогда щелкните правой кнопкой мыши на узле решения в окне Solution Explorer, выберите в контекстном меню пункт Add⇨New Project (Добавить⇨`ClassLibrary1`. В результате создается проект `ClassLibrary1` и добавляется к решению. Далее добавьте ссылку на новый проект, щелкнув правой кнопкой мыши на имени проекта `DefaultAppDomainApp` и выбрав в контекстном меню пункт Add⇨References (Добавить⇨Projects⇨Solution (Проекты⇨`ClassLibrary1`, как показано на рис. 14.3.

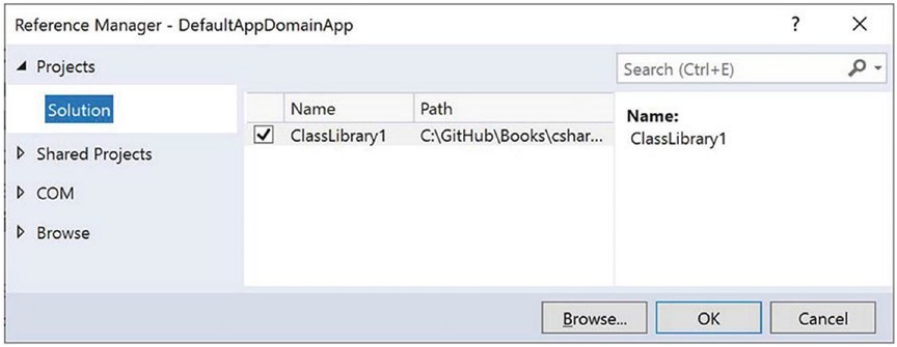

Рис. 14.3. Добавление ссылки на проект в Visual Studio

Добавьте в новую библиотеку классов класс `Car` с таким кодом:

```
namespace ClassLibrary1
{
  public class Car
  {
    public string PetName { get; set; }
    public string Make { get; set; }
    public int Speed { get; set; }
  }
}
```

Теперь, имея новую сборку, добавьте необходимые операторы `using`:

```
using System.IO;
using System.Runtime.Loader;
```

Метод, добавляемый следующим, требует наличия операторов `using` для пространств имен `System.IO` и `System.Runtime.Loader`, которые вы уже добавили в `Program.cs`. Вот код этого метода:

```
static void LoadAdditionalAssembliesDifferentContexts()
{
  var path =
    Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
      "ClassLibrary1.dll");
  AssemblyLoadContext lc1 =
    new AssemblyLoadContext("NewContext1",false);
  var cl1 = lc1.LoadFromAssemblyPath(path);
  var c1 = cl1.CreateInstance("ClassLibrary1.Car");

  AssemblyLoadContext lc2 =
    new AssemblyLoadContext("NewContext2",false);
  var cl2 = lc2.LoadFromAssemblyPath(path);
  var c2 = cl2.CreateInstance("ClassLibrary1.Car");
  Console.WriteLine("*** Loading Additional Assemblies in Different
Contexts ***");
  Console.WriteLine($"Assembly1 Equals(Assembly2) {cl1.Equals(cl2)}");
  Console.WriteLine($"Assembly1 == Assembly2 {cl1 == cl2}");
  Console.WriteLine($"Class1.Equals(Class2) {c1.Equals(c2)}");
  Console.WriteLine($"Class1 == Class2 {c1 == c2}");
}
```

В первой строке кода с применением статического метода `Path.Combine()` строится каталог для сборки `ClassLibrary1`.

На заметку! Вас может интересовать, по какой причине создавалась ссылка на сборку, которая будет загружаться динамически. Это нужно для того, чтобы при компиляции проекта сборка `ClassLibrary1` тоже компилировалась и помещалась в тот же каталог, что и `DefaultAppDomainApp`. В данном примере поступать так попросту удобно. Ссылаться на сборку, которая будет загружаться динамически, нет никакой необходимости.

Далее в коде создается объект `AssemblyLoadContext`, имеющий имя `NewContext1` (первый параметр конструктора) и не поддерживающий выгрузку (второй параметр), который будет использоваться для загрузки сборки `ClassLibrary1` и последующего создания экземпляра класса `Car`. Если какие-то фрагменты кода выглядят для вас незнакомыми, то они будут подробно объясняться в главе 19. Процесс повторяется для еще одного объекта `AssemblyLoadContext`, после чего сборки и классы сравниваются на предмет эквивалентности. В результате выполнения метода `LoadAdditionalAssembliesDifferentContexts()` вы получите следующий вывод:

```
*** Loading Additional Assemblies in Different Contexts ***
Assembly1 Equals(Assembly2) False
Assembly1 == Assembly2 False
Class1.Equals(Class2) False
Class1 == Class2 False
```

Вывод демонстрирует, что та же самая сборка была дважды загружена в домен приложения. Как и следовало ожидать, классы тоже отличаются.

Добавьте новый метод, который будет загружать сборку из того же самого объекта `AssemblyLoadContext`:

```
static void LoadAdditionalAssembliesSameContext()
{
  var path =
    Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
                 "ClassLibrary1.dll");
  AssemblyLoadContext lc1 =
    new AssemblyLoadContext(null,false);
  var cl1 = lc1.LoadFromAssemblyPath(path);
  var c1 = cl1.CreateInstance("ClassLibrary1.Car");
  var cl2 = lc1.LoadFromAssemblyPath(path);
  var c2 = cl2.CreateInstance("ClassLibrary1.Car");
  Console.WriteLine("*** Loading Additional Assemblies in Same Context ***");
  Console.WriteLine($"Assembly1.Equals(Assembly2) {cl1.Equals(cl2)}");
  Console.WriteLine($"Assembly1 == Assembly2 {cl1 == cl2}");
  Console.WriteLine($"Class1.Equals(Class2) {c1.Equals(c2)}");
  Console.WriteLine($"Class1 == Class2 {c1 == c2}");
}
```

Главное отличие приведенного выше кода в том, что создается только один объект `AssemblyLoadContext`. В таком случае, если сборка `ClassLibrary1` загружается дважды, то второй экземпляр сборки является просто указателем на ее первый экземпляр. Выполнение кода дает следующий вывод:

```
*** Loading Additional Assemblies in Same Context ***
Assembly1.Equals(Assembly2) True
Assembly1 == Assembly2 True
Class1.Equals(Class2) False
Class1 == Class2 False
```

Итоговые сведения о процессах, доменах приложений и контекстах загрузки

К настоящему времени вы должны иметь намного лучшее представление о том, как сборка .NET Core обслуживается исполняющей средой. Если изложенный материал показался слишком низкоуровневым, то не переживайте. По большей части .NET Core самостоятельно занимается всеми деталями процессов, доменов приложений и контекстов загрузки. Однако эта информация формирует хороший фундамент для понимания многопоточного программирования на платформе .NET Core.

Резюме

Задачей главы было исследование особенностей обслуживания приложения .NET Core платформой .NET Core. Как вы видели, давно существующее понятие процесса Windows было внутренне изменено и адаптировано под потребности среды CoreCLR. Одиночный процесс (которым можно программно манипулировать посредством типа `System.Diagnostics.Process`) теперь состоит из домена приложения, которое представляет изолированные и независимые границы внутри процесса.

Домен приложения способен размещать и выполнять любое количество связанных сборок. Кроме того, один домен приложения может содержать любое количество контекстов загрузки для дальнейшей изоляции сборок. Благодаря такому дополнительному уровню изоляции типов среда CoreCLR обеспечивает надлежащую обработку объектов с особыми потребностями во время выполнения.

ГЛАВА 15

Многопоточное, параллельное и асинхронное программирование

Вряд ли кому-то понравится работать с приложением, которое притормаживает во время выполнения. Аналогично никто не будет в восторге от того, что запуск какой-то задачи внутри приложения (возможно, по щелчку на элементе в панели инструментов) снижает отзывчивость других частей приложения. До выхода платформы .NET (и .NET Core) построение приложений, способных выполнять сразу несколько задач, обычно требовало написания сложного кода на языке C++, в котором использовались API-интерфейсы многопоточности Windows. К счастью, платформы .NET и .NET Core предлагают ряд способов построения программного обеспечения, которое может совершать нетривиальные операции по уникальным путям выполнения, с намного меньшими сложностями.

Глава начинается с определения общей природы "многопоточного приложения". Затем будет представлено первоначальное пространство имен для многопоточности, поставляемое со времен версии .NET 1.0 и называемое `System.Threading`. Вы ознакомитесь с многочисленными типами (`Thread`, `ThreadStart` и т.д.), которые позволяют явно создавать дополнительные потоки выполнения и синхронизировать разделяемые ресурсы, обеспечивая совместное использование данных несколькими потоками в неизменчивой манере.

В оставшихся разделах главы будут рассматриваться три более новых технологии, которые разработчики приложений .NET Core могут применять для построения многопоточного программного обеспечения: библиотека параллельных задач (Task Parallel Library — TPL), технология PLINQ (Parallel LINQ — параллельный LINQ) и появившиеся относительно недавно (в версии C# 6) ключевые слова, связанные с асинхронной обработкой (`async` и `await`). Вы увидите, что указанные средства помогают значительно упростить процесс создания отзывчивых многопоточных программных приложений.

Отношения между процессом, доменом приложения, контекстом и потоком

В главе 14 *поток* определялся как путь выполнения внутри исполняемого приложения. Хотя многие приложения .NET Core могут успешно и продуктивно работать, будучи однопоточными, первичный поток сборки (создаваемый исполняющей средой при выполнении точки входа приложения) в любое время может порождать вторичные потоки для выполнения дополнительных единиц работы. За счет создания дополнительных потоков можно строить более отзывчивые (но не обязательно быстрее выполняющиеся на одноядерных машинах) приложения.

Пространство имен `System.Threading` появилось в версии .NET 1.0 и предлагает один из подходов к построению многопоточных приложений. Главным типом в этом пространстве имен можно назвать, пожалуй, класс `Thread`, поскольку он представляет отдельный поток. Если необходимо программно получить ссылку на поток, который в текущий момент выполняет заданный член, то нужно просто обратиться к статическому свойству `Thread.CurrentThread`:

```
static void ExtractExecutingThread()
{
  // Получить поток, который в настоящий момент выполняет данный метод.
  Thread currThread = Thread.CurrentThread;
}
```

Вспомните, что в .NET Core существует только один домен приложения. Хотя создавать дополнительные домены приложений нельзя, домен приложения может иметь многочисленные потоки, выполняющиеся в каждый конкретный момент времени. Чтобы получить ссылку на домен приложения, который обслуживает приложение, понадобится вызвать статический метод `Thread.GetDomain()`:

```
static void ExtractAppDomainHostingThread()
{
  // Получить домен приложения, обслуживающий текущий поток.
  AppDomain ad = Thread.GetDomain();
}
```

Одиночный поток в любой момент также может быть перенесен в контекст выполнения и перемещаться внутри нового контекста выполнения по прихоти среды .NET Core Runtime. Для получения текущего контекста выполнения, в котором выполняется поток, используется статическое свойство `Thread.CurrentThread.ExecutionContext`:

```
static void ExtractCurrentThreadExecutionContext()
{
  // Получить контекст выполнения, в котором работает текущий поток.
  ExecutionContext ctx =
    Thread.CurrentThread.ExecutionContext;
}
```

Еще раз: за перемещение потоков в контекст выполнения и из него отвечает среда .NET Core Runtime. Как разработчик приложений .NET Core, вы всегда остаетесь в блаженном неведении относительно того, где завершается каждый конкретный поток. Тем не менее, вы должны быть осведомлены о разнообразных способах получения лежащих в основе примитивов.

Сложность, связанная с параллелизмом

Один из многих болезненных аспектов многопоточного программирования связан с ограниченным контролем над тем, как операционная система или исполняющая среда задействует потоки. Например, написав блок кода, который создает новый поток выполнения, нельзя гарантировать, что этот поток запустится немедленно. Взамен такой код только инструктирует операционную систему или исполняющую среду о необходимости как можно более скорого запуска потока (что обычно происходит, когда планировщик потоков добирается до него).

Кроме того, учитывая, что потоки могут перемещаться между границами приложений и контекстов, как требуется исполняющей среде, вы должны представлять, какие аспекты приложения являются *изменчивыми в потоках* (например, подвергаются многопоточному доступу), а какие операции считаются *атомарными* (операции, изменчивые в потоках, опасны).

Чтобы проиллюстрировать проблему, давайте предположим, что поток вызывает метод специфичного объекта. Теперь представим, что поток приостановлен планировщиком потока, чтобы позволить другому потоку обратиться к тому же методу того же самого объекта.

Если исходный поток не завершил свою операцию, тогда второй входящий поток может увидеть объект в частично модифицированном состоянии. В таком случае второй поток по существу читает фиктивные данные, что определенно может привести к очень странным (и трудно обнаруживаемым) ошибкам, которые еще труднее воспроизвести и устранить.

С другой стороны, атомарные операции в многопоточной среде всегда безопасны. К сожалению, в библиотеках базовых классов .NET Core есть лишь несколько гарантированно атомарных операций. Даже действие по присваиванию значения переменной-члену не является атомарным! Если только в документации по .NET Core специально не сказано об атомарности операции, то вы обязаны считать ее изменчивой в потоках и предпринимать соответствующие меры предосторожности.

Роль синхронизации потоков

К настоящему моменту должно быть ясно, что многопоточные программы сами по себе довольно изменчивы, т.к. многочисленные потоки могут оперировать разделяемыми ресурсами (более или менее) одновременно. Чтобы защитить ресурсы приложений от возможного повреждения, разработчики приложений .NET Core должны применять потоковые примитивы (такие как блокировки, мониторы, атрибут `[Synchronization]` или поддержка языковых ключевых слов) для управления доступом между выполняющимися потоками.

Несмотря на то что платформа .NET Core не способна полностью скрыть сложности, связанные с построением надежных многопоточных приложений, сам процесс был значительно упрощен. Используя типы из пространства имен `System.Threading`, библиотеку TPL и ключевые слова `async` и `await` языка C#, можно работать с множеством потоков, прикладывая минимальные усилия.

Прежде чем погрузиться в детали пространства имен `System.Threading`, библиотеки TPL и ключевых слов `async` и `await` языка C#, мы начнем с выяснения того, каким образом можно применять тип делегата .NET Core для вызова метода в асинхронной манере. Хотя вполне справедливо утверждать, что с выходом версии .NET 4.6 ключевые слова `async` и `await` предлагают более простую альтернативу асинхронным делегатам, по-прежнему важно знать способы взаимодействия с кодом, исполь-

зующим этот подход (в производственной среде имеется масса кода, в котором применяются асинхронные делегаты).

Пространство имен System.Threading

В рамках платформ .NET и .NET Core пространство имен `System.Threading` предоставляет типы, которые дают возможность напрямую конструировать многопоточные приложения. В дополнение к типам, позволяющим взаимодействовать с потоком .NET Core Runtime, в `System.Threading` определены типы, которые открывают доступ к пулу потоков, обслуживаемому .NET Core Runtime, простому (не связанному с графическим пользовательским интерфейсом) классу `Timer` и многочисленным типам, применяемым для синхронизированного доступа к разделяемым ресурсам.

В табл. 15.1 перечислены некоторые важные члены пространства имен `System.Threading`. (За полными сведениями обращайтесь в документацию по .NET Core.)

Таблица 15.1. Основные типы пространства имен `System.Threading`

Тип	Назначение
`Interlocked`	Этот тип предоставляет атомарные операции для переменных, разделяемых между несколькими потоками
`Monitor`	Этот тип обеспечивает синхронизацию потоковых объектов, используя блокировки и ожидания/сигналы. Ключевое слово `lock` языка C# "за кулисами" применяет объект `Monitor`
`Mutex`	Этот примитив синхронизации может использоваться для синхронизации между границами доменов приложений
`ParameterizedThreadStart`	Этот делегат позволяет потоку вызывать методы, принимающие произвольное количество аргументов
`Semaphore`	Этот тип позволяет ограничивать количество потоков, которые могут иметь доступ к ресурсу или к определенному типу ресурсов одновременно
`Thread`	Этот тип представляет поток, выполняемый в среде .NET Core Runtime. С применением данного типа можно порождать дополнительные потоки в исходном домене приложения
`ThreadPool`	Этот тип позволяет взаимодействовать с поддерживаемым средой .NET Core Runtime пулом потоков внутри заданного процесса
`ThreadPriority`	Это перечисление представляет уровень приоритета потока (`Highest`, `Normal` и т.д.)
`ThreadStart`	Этот делегат позволяет указывать метод для вызова в заданном потоке. В отличие от делегата `ParametrizedThreadStart` целевые методы `ThreadStart` всегда должны иметь один и тот же прототип
`ThreadState`	Это перечисление задает допустимые состояния потока (`Running`, `Aborted` и т.д.)
`Timer`	Этот тип предоставляет механизм выполнения метода через указанные интервалы
`TimerCallback`	Этот тип делегата используется в сочетании с типами `Timer`

Класс System.Threading.Thread

Класс Thread является самым элементарным из всех типов в пространстве имен System.Threading. Он представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри отдельного домена приложения. В этом классе определено несколько методов (статических и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего домена приложения, а также приостанавливать, останавливать и уничтожать указанный поток. Список основных статических членов приведен в табл. 15.2.

Таблица 15.2. Основные статические члены типа Thread

Статический член	Назначение
ExecutionContext	Это свойство только для чтения возвращает информацию, касающуюся логического потока выполнения, куда входят контексты безопасности, вызова, синхронизации, локализации и транзакции
CurrentThread	Это свойство только для чтения возвращает ссылку на текущий выполняемый поток
Sleep()	Этот метод приостанавливает текущий поток на указанное время

Класс Thread также поддерживает члены уровня экземпляра, часть которых описана в табл. 15.3.

Таблица 15.3. Избранные члены уровня экземпляра типа Thread

Член уровня экземпляра	Назначение
IsAlive	Возвращает булевское значение, которое указывает, были ли запущен поток (и пока еще не прекращен или не отменен)
IsBackground	Получает или устанавливает значение, которое указывает, является ли данный поток фоновым (что более подробно объясняется далее в главе)
Name	Позволяет установить дружественное текстовое имя потока
Priority	Получает или устанавливает приоритет потока, который может принимать значение из перечисления ThreadPriority
ThreadState	Получает состояние данного потока, которое может принимать значение из перечисления ThreadState
Abort()	Указывает среде .NET Core Runtime на необходимость как можно более скорого прекращения работы потока
Interrupt()	Прерывает (например, приостанавливает) текущий поток на подходящий период ожидания
Join()	Блокирует вызывающий поток до тех пор, пока указанный поток (тот, на котором вызван метод Join()) не завершится
Resume()	Возобновляет выполнение ранее приостановленного потока
Start()	Указывает среде .NET Core Runtime на необходимость как можно более скорого запуска потока
Suspend()	Приостанавливает поток. Если поток уже приостановлен, то вызов Suspend() не оказывает никакого действия

На заметку! Прекращение работы или приостановка активного потока обычно считается плохой идеей. В таком случае есть шанс (хотя и небольшой), что поток может допустить "утечку" своей рабочей нагрузки.

Получение статистических данных о текущем потоке выполнения

Вспомните, что точка входа исполняемой сборки (т.е. операторы верхнего уровня или метод `Main()`) запускается в первичном потоке выполнения. Чтобы проиллюстрировать базовое применение типа `Thread`, предположим, что имеется новый проект консольного приложения по имени `ThreadStats`. Как вам известно, статическое свойство `Thread.CurrentThread` извлекает объект `Thread`, который представляет поток, выполняющийся в текущий момент. Получив текущий поток, можно вывести разнообразные статистические сведения о нем:

```
// Не забудьте импортировать пространство имен System.Threading.
using System;
using System.Threading;

Console.WriteLine("***** Primary Thread stats *****\n");

// Получить имя текущего потока.
Thread primaryThread = Thread.CurrentThread;
primaryThread.Name = "ThePrimaryThread";

// Вывести статистические данные о текущем потоке.
Console.WriteLine("ID of current thread: {0}",
  Thread.CurrentContext.ContextID);   // Идентификатор текущего потока
Console.WriteLine("Thread Name: {0}",
  primaryThread.Name);                // Имя потока
Console.WriteLine("Has thread started?: {0}",
  primaryThread.IsAlive);             // Запущен ли поток
Console.WriteLine("Priority Level: {0}",
  primaryThread.Priority);            // Приоритет потока
Console.WriteLine("Thread State: {0}",
  primaryThread.ThreadState);         // Состояние потока
Console.ReadLine();
```

Вот как выглядит вывод:

```
***** Primary Thread stats *****

ID of current thread: 1
Thread Name: ThePrimaryThread
Has thread started?: True
Priority Level: Normal
Thread State: Running
```

Свойство Name

Обратите внимание, что класс `Thread` поддерживает свойство по имени `Name`. Если значение `Name` не было установлено, тогда будет возвращаться пустая строка. Однако назначение конкретному объекту `Thread` дружественного имени может значительно упростить отладку. Во время сеанса отладки в Visual Studio можно открыть окно Threads (Потоки), выбрав пункт меню Debug⇨Windows⇨Threads (Отладка⇨Окна⇨Потоки). На рис. 15.1 легко заметить, что окно Threads позволяет быстро идентифицировать поток, который нужно диагностировать.

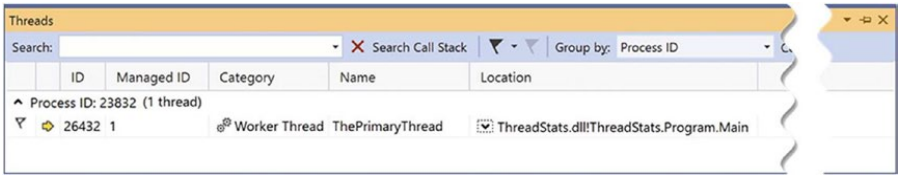

Рис. 15.1. Отладка потока в Visual Studio

Свойство `Priority`

Далее обратите внимание, что в типе `Thread` определено свойство по имени `Priority`. По умолчанию все потоки имеют уровень приоритета `Normal`. Тем не менее, в любой момент жизненного цикла потока его можно изменить, используя свойство `Priority` и связанное с ним перечисление `System.Threading.ThreadPriority`:

```
public enum ThreadPriority
{
  Lowest,
  BelowNormal,
  Normal,        // Стандартное значение.
  AboveNormal,
  Highest
}
```

В случае присваивания уровню приоритета потока значения, отличающегося от стандартного (`ThreadPriority.Normal`), помните об отсутствии прямого контроля над тем, когда планировщик потоков будет переключать потоки между собой. Уровень приоритета потока предоставляет среде .NET Core Runtime лишь подсказку относительно важности действия потока. Таким образом, поток с уровнем приоритета `ThreadPriority.Highest` не обязательно гарантированно получит наивысший приоритет.

Опять-таки, если планировщик потоков занят решением определенной задачи (например, синхронизацией объекта, переключением потоков либо их перемещением), то уровень приоритета, скорее всего, будет соответствующим образом изменен. Однако при прочих равных условиях среда .NET Core Runtime прочитает эти значения и проинструктирует планировщик потоков о том, как лучше выделять кванты времени. Потоки с идентичными уровнями приоритета должны получать одинаковое количество времени на выполнение своей работы.

В большинстве случаев необходимость в прямом изменении уровня приоритета потока возникает редко (если вообще возникает). Теоретически можно так повысить уровень приоритета набора потоков, что в итоге воспрепятствовать выполнению низкоприоритетных потоков с их запрошенными уровнями (поэтому соблюдайте осторожность).

Ручное создание вторичных потоков

Когда вы хотите программно создать дополнительные потоки для выполнения какой-то единицы работы, то во время применения типов из пространства имен `System.Threading` следуйте представленному ниже предсказуемому процессу.

1. Создать метод, который будет служить точкой входа для нового потока.
2. Создать новый делегат `ParametrizedThreadStart` (или `ThreadStart`), передав его конструктору адрес метода, который был определен на шаге 1.
3. Создать объект `Thread`, передав конструктору в качестве аргумента делегат `ParametrizedThreadStart/ThreadStart`.
4. Установить начальные характеристики потока (имя, приоритет и т.д.).
5. Вызвать метод `Thread.Start()`, что приведет к как можно более скорому запуску потока для метода, на который ссылается делегат, созданный на шаге 2.

Согласно шагу 2 для указания на метод, который будет выполняться во вторичном потоке, можно использовать два разных типа делегата. Делегат `ThreadStart` способен указывать на любой метод, который не принимает аргументов и ничего не возвращает. Такой делегат может быть полезен, когда метод предназначен просто для запуска в фоновом режиме без дальнейшего взаимодействия с ним.

Ограничение `ThreadStart` связано с невозможностью передавать ему параметры для обработки. Тем не менее, тип делегата `ParametrizedThreadStart` позволяет передать единственный параметр типа `System.Object`. Учитывая, что с помощью `System.Object` представляется все, что угодно, посредством специального класса или структуры можно передавать любое количество параметров. Однако имейте в виду, что делегаты `ThreadStart` и `ParametrizedThreadStart` могут указывать только на методы, возвращающие `void`.

Работа с делегатом `ThreadStart`

Чтобы проиллюстрировать процесс построения многопоточного приложения (а также его полезность), создайте проект консольного приложения по имени `SimpleMultiThreadApp`, которое позволит конечному пользователю выбирать, будет приложение выполнять свою работу в единственном первичном потоке или же разделит рабочую нагрузку с применением двух отдельных потоков выполнения.

После импортирования пространства имен `System.Threading` определите метод для выполнения работы (возможного) вторичного потока. Чтобы сосредоточиться на механизме построения многопоточных программ, этот метод будет просто выводить на консоль последовательность чисел, делая на каждом шаге паузу примерно в 2 секунды. Ниже показано полное определение класса `Printer`:

```
using System;
using System.Threading;

namespace SimpleMultiThreadApp
{
  public class Printer
  {
    public void PrintNumbers()
    {
      // Вывести информацию о потоке.
      Console.WriteLine("-> {0} is executing PrintNumbers()",
        Thread.CurrentThread.Name);

      // Вывести числа.
      Console.Write("Your numbers: ");
      for(int i = 0; i < 10; i++)
      {
```

```
        Console.Write("{0}, ", i);
        Thread.Sleep(2000);
      }
      Console.WriteLine();
    }
  }
}
```

Добавьте в файл `Program.cs` операторы верхнего уровня, которые предложат пользователю решить, сколько потоков будет использоваться для выполнения работы приложения: один или два. Если пользователь запрашивает один поток, то нужно просто вызвать метод `PrintNumbers()` в первичном потоке. Тем не менее, когда пользователь выбирает два потока, понадобится создать делегат `ThreadStart`, указывающий на `PrintNumbers()`, передать объект делегата конструктору нового объекта `Thread` и вызвать метод `Start()` для информирования среды .NET Core Runtime о том, что данный поток готов к обработке. Вот полная реализация:

```
using System;
using System.Threading;
using SimpleMultiThreadApp;

Console.WriteLine("***** The Amazing Thread App *****\n");
Console.Write("Do you want [1] or [2] threads? ");
string threadCount = Console.ReadLine();  // Запрос количества потоков

// Назначить имя текущему потоку.
Thread primaryThread = Thread.CurrentThread;
primaryThread.Name = "Primary";

// Вывести информацию о потоке.
Console.WriteLine("-> {0} is executing Main()",
Thread.CurrentThread.Name);

// Создать рабочий класс.
Printer p = new Printer();

switch(threadCount)
{
  case "2":
    // Создать поток.
    Thread backgroundThread =
      new Thread(new ThreadStart(p.PrintNumbers));
    backgroundThread.Name = "Secondary";
    backgroundThread.Start();
    break;
  case "1":
    p.PrintNumbers();
    break;
  default:
    Console.WriteLine("I don't know what you want...you get 1 thread.");
    goto case "1";   // Переход к варианту с одним потоком
}

// Выполнить некоторую дополнительную работу.
MessageBox.Show("I'm busy!", "Work on main thread...");
Console.ReadLine();
```

Если теперь вы запустите программу с одним потоком, то обнаружите, что финальное окно сообщения не будет отображать сообщение, пока вся последовательность чисел не выведется на консоль. Поскольку после вывода каждого числа установлена пауза около 2 секунд, это создаст не особенно приятное впечатление у конечного пользователя. Однако в случае выбора двух потоков окно сообщения отображается немедленно, потому что для вывода чисел на консоль выделен отдельный объект `Thread`.

Работа с делегатом `ParametrizedThreadStart`

Вспомните, что делегат `ThreadStart` может указывать только на методы, которые возвращают `void` и не принимают аргументов. В некоторых случаях это подходит, но если нужно передать данные методу, выполняющемуся во вторичном потоке, тогда придется использовать тип делегата `ParametrizedThreadStart`. В целях иллюстрации создайте новый проект консольного приложения по имени `AddWithThreads` и импортируйте пространство имен `System.Threading`. С учетом того, что делегат `ParametrizedThreadStart` может указывать на любой метод, принимающий параметр типа `System.Object`, постройте специальный тип, который содержит числа, подлежащие сложению:

```
namespace AddWithThreads
{
  class AddParams
  {
    public int a, b;

    public AddParams(int numb1, int numb2)
    {
      a = numb1;
      b = numb2;
    }
  }
}
```

Далее создайте в классе `Program` статический метод, который принимает параметр `AddParams` и выводит на консоль сумму двух чисел:

```
void Add(object data)
{
  if (data is AddParams ap)
  {
    Console.WriteLine("ID of thread in Add(): {0}",
      Thread.CurrentThread.ManagedThreadId);

    Console.WriteLine("{0} + {1} is {2}",
      ap.a, ap.b, ap.a + ap.b);
  }
}
```

Код в файле `Program.cs` прямолинеен. Вместо типа `ThreadStart` просто используется `ParametrizedThreadStart`:

```
using System;
using System.Threading;
using AddWithThreads;

Console.WriteLine("***** Adding with Thread objects *****");
```

```
Console.WriteLine("ID of thread in Main(): {0}",
  Thread.CurrentThread.ManagedThreadId);

// Создать объект AddParams для передачи вторичному потоку.
AddParams ap = new AddParams(10, 10);
Thread t = new Thread(new ParameterizedThreadStart(Add));
t.Start(ap);

// Подождать, пока другой поток завершится.
Thread.Sleep(5);
Console.ReadLine();
```

Класс AutoResetEvent

В приведенных выше начальных примерах нет какого-либо надежного способа узнать, когда вторичный поток завершит свою работу. В последнем примере метод `Sleep()` вызывался с произвольным временным периодом, чтобы дать возможность другому потоку завершиться. Простой и безопасный к потокам способ заставить один поток ожидать, пока не завершится другой поток, предусматривает применение класса `AutoResetEvent`. В потоке, который должен ожидать, создайте экземпляр `AutoResetEvent` и передайте его конструктору значение `false`, указав, что уведомления пока не было. Затем в точке, где требуется ожидать, вызовите метод `WaitOne()`. Ниже приведен модифицированный класс `Program`, который делает все описанное с использованием статической переменной-члена `AutoResetEvent`:

```
AutoResetEvent _waitHandle = new AutoResetEvent(false);

Console.WriteLine("***** Adding with Thread objects *****");
Console.WriteLine("ID of thread in Main(): {0}",
  Thread.CurrentThread.ManagedThreadId);
AddParams ap = new AddParams(10, 10);
Thread t = new Thread(new ParameterizedThreadStart(Add));
t.Start(ap);

// Ожидать, пока не поступит уведомление!
_waitHandle.WaitOne();
Console.WriteLine("Other thread is done!");

Console.ReadLine();
...
```

Когда другой поток завершит свою работу, он вызовет метод `Set()` на том же самом экземпляре типа `AutoResetEvent`:

```
void Add(object data)
{
  if (data is AddParams ap)
  {
    Console.WriteLine("ID of thread in Add(): {0}",
      Thread.CurrentThread.ManagedThreadId);

    Console.WriteLine("{0} + {1} is {2}",
      ap.a, ap.b, ap.a + ap.b);

    // Сообщить другому потоку о том, что работа завершена.
    _waitHandle.Set();
  }
}
```

Потоки переднего плана и фоновые потоки

Теперь, когда вы знаете, как программно создавать новые потоки выполнения с применением типов из пространства имен `System.Threading`, давайте формализуем разницу между потоками переднего плана и фоновыми потоками.

- *Потоки переднего плана* имеют возможность предохранять текущее приложение от завершения. Среда .NET Core Runtime не будет прекращать работу приложения (скажем, выгружая обслуживающий домен приложения) до тех пор, пока не будут завершены все потоки переднего плана.
- *Фоновые потоки* (иногда называемые *потоками-демонами*) воспринимаются средой .NET Core Runtime как расширяемые пути выполнения, которые в любой момент времени могут быть проигнорированы (даже если они заняты выполнением некоторой части работы). Таким образом, если при выгрузке домена приложения все потоки переднего плана завершены, то все фоновые потоки автоматически уничтожаются.

Важно отметить, что потоки переднего плана и фоновые потоки — *не* синонимы первичных и рабочих потоков. По умолчанию каждый поток, создаваемый посредством метода `Thread.Start()`, автоматически становится потоком переднего плана. В итоге домен приложения не выгрузится до тех пор, пока все потоки выполнения не завершат свои единицы работы. В большинстве случаев именно такое поведение и требуется.

Ради доказательства сделанных утверждений предположим, что метод `Printer.PrintNumbers()` необходимо вызвать во вторичном потоке, который должен вести себя как фоновый. Это означает, что метод, указываемый типом `Thread` (через делегат `ThreadStart` или `ParametrizedThreadStart`), должен обладать возможностью безопасного останова, как только все потоки переднего плана закончат свою работу. Конфигурирование такого потока сводится просто к установке свойства `IsBackground` в `true`:

```
Console.WriteLine("***** Background Threads *****\n");
Printer p = new Printer();
Thread bgroundThread =
  new Thread(new ThreadStart(p.PrintNumbers));

// Теперь это фоновый поток.
bgroundThread.IsBackground = true;
bgroundThread.Start();
```

Обратите внимание, что в приведенном выше коде *не* делается вызов `Console.ReadLine()`, чтобы заставить окно консоли оставаться видимым, пока не будет нажата клавиша <Enter>. Таким образом, после запуска приложение немедленно прекращается, потому что объект `Thread` сконфигурирован как фоновый поток. С учетом того, что точка входа приложения (приведенные здесь операторы верхнего уровня или метод `Main()`) инициирует создание первичного потока *переднего плана*, как только логика в точке входа завершится, домен приложения будет выгружен, прежде чем вторичный поток сможет закончить свою работу.

Однако если закомментировать строку, которая устанавливает свойство `IsBackground` в `true`, то обнаружится, что на консоль выводятся все числа, поскольку все потоки переднего плана должны завершить свою работу перед тем, как домен приложения будет выгружен из обслуживающего процесса.

По большей части конфигурировать поток для функционирования в фоновом режиме может быть удобно, когда интересующий рабочий поток выполняет некритичную задачу, потребность в которой исчезает после завершения главной задачи программы. Например, можно было бы построить приложение, которое проверяет почтовый сервер каждые несколько минут на предмет поступления новых сообщений электронной почты, обновляет текущий прогноз погоды или решает какие-то другие некритичные задачи.

Проблема параллелизма

При построении многопоточных приложений необходимо гарантировать, что любой фрагмент разделяемых данных защищен от возможности изменения со стороны сразу нескольких потоков. Поскольку все потоки в домене приложения имеют параллельный доступ к разделяемым данным приложения, вообразите, что может произойти, если множество потоков одновременно обратятся к одному и тому же элементу данных. Так как планировщик потоков случайным образом приостанавливает их работу, что если поток A будет вытеснен до завершения своей работы? Тогда поток B прочитает нестабильные данные.

Чтобы проиллюстрировать проблему, связанную с параллелизмом, давайте создадим еще один проект консольного приложения под названием `MultiThreadedPrinting`. В приложении снова будет использоваться построенный ранее класс `Printer`, но на этот раз метод `PrintNumbers()` приостановит текущий поток на сгенерированный случайным образом период времени.

```
using System;
using System.Threading;

namespace MultiThreadedPrinting
{
  public class Printer
  {
    public void PrintNumbers()
    {
      // Отобразить информацию о потоке.
      Console.WriteLine("-> {0} is executing PrintNumbers()",
        Thread.CurrentThread.Name);

      // Вывести числа.
      for (int i = 0; i < 10; i++)
      {
        // Приостановить поток на случайный период времени.
        Random r = new Random();
        Thread.Sleep(1000 * r.Next(5));
        Console.Write("{0}, ", i);
      }
      Console.WriteLine();
    }
  }
}
```

Вызывающий код отвечает за создание массива из десяти (уникально именованных) объектов `Thread`, каждый из которых вызывает метод *одного и того же экземпляра* класса `Printer`:

```
using System;
using System.Threading;
using MultiThreadedPrinting;
Console.WriteLine("*****Synchronizing Threads *****\n");
Printer p = new Printer();
// Создать 10 потоков, которые указывают на один
// и тот же метод того же самого объекта.
Thread[] threads = new Thread[10];
for (int i = 0; i < 10; i++)
{
  threads[i] = new Thread(new ThreadStart(p.PrintNumbers))
  {
    Name = $"Worker thread #{i}"
  };
}
// Теперь запустить их все.
foreach (Thread t in threads)
{
  t.Start();
}
Console.ReadLine();
```

Прежде чем взглянуть на тестовые запуски, кратко повторим суть проблемы. Первичный поток внутри этого домена приложения начинает свое существование с порождения десяти вторичных рабочих потоков. Каждому рабочему потоку указывается на необходимость вызова метода `PrintNumbers()` *того же самого* экземпляра класса `Printer`. Поскольку никаких мер для блокировки разделяемых ресурсов данного объекта (консоли) не предпринималось, есть неплохой шанс, что текущий поток будет вытеснен до того, как метод `PrintNumbers()` выведет полные результаты. Из-за того, что не известно в точности, когда подобное может произойти (если вообще произойдет), будут получены непредсказуемые результаты. Например, вывод может выглядеть так:

```
*****Synchronizing Threads *****
-> Worker thread #3 is executing PrintNumbers()
-> Worker thread #0 is executing PrintNumbers()
-> Worker thread #1 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
-> Worker thread #4 is executing PrintNumbers()
-> Worker thread #5 is executing PrintNumbers()
-> Worker thread #6 is executing PrintNumbers()
-> Worker thread #7 is executing PrintNumbers()
-> Worker thread #8 is executing PrintNumbers()
-> Worker thread #9 is executing PrintNumbers()
0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 2, 3, 1, 2, 2, 2, 1, 2,
1, 1, 2, 2, 3, 3, 4,
3, 3, 2, 2, 3, 4, 3, 4, 5, 4, 5, 4, 4, 3, 6, 7, 2, 3, 4, 4, 4, 5, 6, 5,
3, 5, 8, 9,
6, 7, 4, 5, 6, 6, 5, 5, 5, 8, 5, 6, 7, 8, 7, 7, 6, 6, 6, 8, 9,
8, 7, 7, 7, 7, 9,
6, 8, 9,
8, 9,
9, 9,
8, 8, 7, 8, 9,
9,
9,
```

Запустите приложение еще несколько раз. Скорее всего, каждый раз вы будете получать отличающийся вывод.

На заметку! Если получить непредсказуемый вывод не удается, увеличьте количество потоков с 10 до 100 (например) или добавьте в код еще один вызов `Thread.Sleep()`. В конце концов, вы столкнетесь с проблемой параллелизма.

Должно быть совершенно ясно, что здесь присутствуют проблемы. В то время как каждый поток сообщает экземпляру `Printer` о необходимости вывода числовых данных, планировщик потоков благополучно переключает потоки в фоновом режиме. В итоге получается несогласованный вывод. Нужен способ программной реализации синхронизированного доступа к разделяемым ресурсам. Как и можно было предположить, пространство имен `System.Threading` предлагает несколько типов, связанных с синхронизацией. В языке C# также предусмотрено ключевое слово для синхронизации разделяемых данных в многопоточных приложениях.

Синхронизация с использованием ключевого слова `lock` языка C#

Первый прием, который можно применять для синхронизации доступа к разделяемым ресурсам, предполагает использование ключевого слова `lock` языка C#. Оно позволяет определять блок операторов, которые должны быть синхронизованными между потоками. В результате входящие потоки не могут прерывать текущий поток, мешая ему завершить свою работу. Ключевое слово `lock` требует указания *маркера* (объектной ссылки), который должен быть получен потоком для входа в область действия блокировки. Чтобы попытаться заблокировать *закрытый* метод уровня экземпляра, необходимо просто передать ссылку на текущий тип:

```
private void SomePrivateMethod()
{
  // Использовать текущий объект как маркер потока.
  lock(this)
  {
    // Весь код внутри этого блока является безопасным к потокам.
  }
}
```

Тем не менее, если блокируется область кода внутри *открытого* члена, то безопаснее (да и рекомендуется) объявить закрытую переменную-член типа `object` для применения в качестве маркера блокировки:

```
public class Printer
{
  // Маркер блокировки.
  private object threadLock = new object();

  public void PrintNumbers()
  {
    // Использовать маркер блокировки.
    lock (threadLock)
    {
      ...
    }
  }
}
```

В любом случае, если взглянуть на метод `PrintNumbers()`, то можно заметить, что разделяемым ресурсом, за доступ к которому соперничают потоки, является окно консоли. Поместите весь код взаимодействия с типом `Console` внутрь области `lock`, как показано ниже:

```
public void PrintNumbers()
{
  // Использовать в качестве маркера блокировки закрытый член object.
  lock (threadLock)
  {
    // Вывести информацию о потоке.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);
    // Вывести числа.
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
    Console.WriteLine();
  }
}
```

В итоге вы построили метод, который позволит текущему потоку завершить свою задачу. Как только поток входит в область `lock`, маркер блокировки (в данном случае ссылка на текущий объект) становится недоступным другим потокам до тех пор, пока блокировка не будет освобождена после выхода из области `lock`. Таким образом, если поток A получил маркер блокировки, то другие потоки не смогут войти *ни в одну из областей*, которые используют тот же самый маркер, до тех пор, пока поток A не освободит его.

На заметку! Если необходимо блокировать код в статическом методе, тогда следует просто объявить закрытую статическую переменную-член типа `object`, которая и будет служить маркером блокировки.

Запустив приложение, вы заметите, что каждый поток получил возможность выполнить свою работу до конца:

```
*****Synchronizing Threads *****
-> Worker thread #0 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #1 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #2 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #4 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #5 is executing PrintNumbers()
```

```
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #7 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #6 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #8 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #9 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Синхронизация с использованием типа `System.Threading.Monitor`

Оператор `lock` языка C# на самом деле представляет собой сокращение для работы с классом `System.Threading.Monitor`. При обработке компилятором C# область `lock` преобразуется в следующую конструкцию (в чем легко убедиться с помощью утилиты `ldasm.exe`):

```
public void PrintNumbers()
{
  Monitor.Enter(threadLock);
  try
  {
    // Вывести информацию о потоке.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);
    // Вывести числа.
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
    Console.WriteLine();
  }
  finally
  {
    Monitor.Exit(threadLock);
  }
}
```

Первым делом обратите внимание, что конечным получателем маркера потока, который указывается как аргумент ключевого слова `lock`, является метод `Monitor.Enter()`. Весь код внутри области `lock` помещен внутрь блока `try`. Соответствующий блок `finally` гарантирует освобождение маркера блокировки (посредством метода `Monitor.Exit()`), даже если возникнут любые исключения времени выполнения. Модифицировав программу `MultiThreadShareData` с целью прямого применения типа `Monitor` (как только что было показано), вы обнаружите, что вывод идентичен.

С учетом того, что ключевое слово `lock` требует написания меньшего объема кода, чем при явной работе с типом `System.Threading.Monitor`, может возникнуть вопрос о преимуществах использования этого типа напрямую. Выражаясь кратко, тип `Monitor` обеспечивает большую степень контроля. Применяя тип `Monitor`, можно

заставить активный поток ожидать в течение некоторого периода времени (с помощью статического метода `Monitor.Wait()`), информировать ожидающие потоки о том, что текущий поток завершен (через статические методы `Monitor.Pulse()` и `Monitor.PulseAll()`), и т.д.

Как и можно было ожидать, в значительном числе случаев ключевого слова `lock` будет достаточно. Если вас интересуют дополнительные члены класса `Monitor`, тогда обращайтесь в документацию по .NET Core.

Синхронизация с использованием типа `System.Threading.Interlocked`

Не заглядывая в код CIL, обычно нелегко поверить в то, что присваивание и простые арифметические операции *не являются атомарными*. По указанной причине в пространстве имен `System.Threading` предоставляется тип, который позволяет атомарно оперировать одиночным элементом данных с меньшими накладными расходами, чем тип `Monitor`. В классе `Interlocked` определены статические члены, часть которых описана в табл. 15.4.

Таблица 15.4. Избранные статические члены типа `System.Threading.Interlocked`

Член	Назначение
`CompareExchange()`	Безопасно проверяет два значения на равенство и, если они равны, то заменяет одно из значений третьим
`Decrement()`	Безопасно уменьшает значение на 1
`Exchange()`	Безопасно меняет два значения местами
`Increment()`	Безопасно увеличивает значение на 1

Несмотря на то что это не сразу видно, процесс атомарного изменения одиночного значения довольно часто применяется в многопоточной среде. Пусть имеется код, который инкрементирует целочисленную переменную-член по имени `intVal`. Вместо написания кода синхронизации вроде показанного ниже:

```
int intVal = 5;
object myLockToken = new();
lock(myLockToken)
{
  intVal++;
}
```

код можно упростить, используя статический метод `Interlocked.Increment()`. Методу потребуется передать инкрементируемую переменную по ссылке. Обратите внимание, что метод `Increment()` не только изменяет значение входного параметра, но также возвращает полученное новое значение:

```
intVal = Interlocked.Increment(ref intVal);
```

В дополнение к методам `Increment()` и `Decrement()` тип `Interlocked` позволяет атомарно присваивать числовые и объектные данные. Например, чтобы присвоить переменной-члену значение `83`, можно обойтись без явного оператора `lock` (или явной логики `Monitor`) и применить метод `Interlock.Exchange()`:

```
Interlocked.Exchange(ref myInt, 83);
```

Наконец, если необходимо проверить два значения на предмет равенства и изменить элемент сравнения в безопасной к потокам манере, тогда допускается использовать метод `Interlocked.CompareExchange()`:

```
public void CompareAndExchange()
{
  // Если значение i равно 83, то изменить его на 99.
  Interlocked.CompareExchange(ref i, 99, 83);
}
```

Программирование с использованием обратных вызовов `Timer`

Многие приложения нуждаются в вызове специфического метода через регулярные интервалы времени. Например, в приложении может существовать необходимость в отображении текущего времени внутри панели состояния с помощью определенной вспомогательной функции. Или, скажем, нужно, чтобы приложение эпизодически вызывало вспомогательную функцию, выполняющую некритичные фоновые задачи, такие как проверка поступления новых сообщений электронной почты. В ситуациях подобного рода можно применять тип `System.Threading.Timer` в сочетании со связанным делегатом по имени `TimerCallback`.

В целях иллюстрации предположим, что у вас есть проект консольного приложения (`TimerApp`), которое будет выводить текущее время каждую секунду до тех пор, пока пользователь не нажмет клавишу <Enter> для прекращения работы приложения. Первый очевидный шаг — написание метода, который будет вызываться типом `Timer` (не забудьте импортировать в свой файл кода пространство имен `System.Threading`):

```
using System;
using System.Threading;

Console.WriteLine("***** Working with Timer type *****\n");
Console.ReadLine();
static void PrintTime(object state)
{
  Console.WriteLine("Time is: {0}",
    DateTime.Now.ToLongTimeString());
}
```

Обратите внимание, что метод `PrintTime()` принимает единственный параметр типа `System.Object` и возвращает `void`. Это обязательно, потому что делегат `TimerCallback` может вызывать только методы, которые соответствуют такой сигнатуре. Значение, передаваемое целевому методу делегата `TimerCallback`, может быть объектом любого типа (в случае примера с электронной почтой параметр может представлять имя сервера Microsoft Exchange Server для взаимодействия в течение процесса). Также обратите внимание, что поскольку параметр на самом деле является экземпляром типа `System.Object`, в нем можно передавать несколько аргументов, используя `System.Array` или специальный класс либо структуру.

Следующий шаг связан с конфигурированием экземпляра делегата `TimerCallback` и передачей его объекту `Timer`. В дополнение к настройке делегата `TimerCallback` конструктор `Timer` позволяет указывать необязательный информационный параметр для передачи целевому методу делегата (определенный как `System.Object`), интер-

вал вызова метода и период ожидания (в миллисекундах), который должен истечь перед первым вызовом. Вот пример:

```
Console.WriteLine("***** Working with Timer type *****\n");

// Создать делегат для типа Timer.
TimerCallback timeCB = new TimerCallback(PrintTime);

// Установить параметры таймера.
Timer t = new Timer(
  timeCB,       // Объект делегата TimerCallback.
  null,         // Информация для передачи в вызванный метод
                // (null, если информация отсутствует).
  0,            // Период ожидания перед запуском (в миллисекундах).
  1000);        // Интервал между вызовами (в миллисекундах).

Console.WriteLine("Hit key to terminate...");
Console.ReadLine();
```

В этом случае метод `PrintTime()` вызывается приблизительно каждую секунду и не получает никакой дополнительной информации. Ниже показан вывод примера:

```
***** Working with Timer type *****
Time is: 6:51:48 PM
Time is: 6:51:49 PM
Time is: 6:51:50 PM
Time is: 6:51:51 PM
Time is: 6:51:52 PM
Hit key to terminate...
```

Чтобы передать целевому методу делегата какую-то информацию, необходимо просто заменить значение `null` во втором параметре конструктора подходящей информацией, например:

```
// Установить параметры таймера.
Timer t = new Timer(timeCB, "Hello From C# 9.0", 0, 1000);
```

Получить входные данные можно следующим образом:

```
static void PrintTime(object state)
{
  Console.WriteLine("Time is: {0}, Param is: {1}",
    DateTime.Now.ToLongTimeString(), state.ToString());
}
```

Использование автономного отбрасывания (нововведение в версии 7.0)

В предыдущем примере переменная `Timer` не применяется в каком-либо пути выполнения и потому может быть заменена отбрасыванием:

```
var _ = new Timer(
  timeCB,       // Объект делегата TimerCallback.
  null,         // Информация для передачи в вызванный метод
                // (null, если информация отсутствует).
  0,            // Период ожидания перед запуском
                // (в миллисекундах).
  1000);        // Интервал между вызовами
                // (в миллисекундах).
```

Класс ThreadPool

Следующей темой о потоках, которую мы рассмотрим в настоящей главе, будет роль пула потоков. Запуск нового потока связан с затратами, поэтому в целях повышения эффективности пул потоков удерживает созданные (но неактивные) потоки до тех пор, пока они не понадобятся. Для взаимодействия с этим пулом ожидающих потоков в пространстве имен `System.Threading` предлагается класс `ThreadPool`.

Чтобы запросить поток из пула для обработки вызова метода, можно использовать метод `ThreadPool.QueueUserWorkItem()`. Он имеет перегруженную версию, которая позволяет в дополнение к экземпляру делегата `WaitCallback` указывать необязательный параметр `System.Object` для передачи специальных данных состояния:

```
public static class ThreadPool
{
  ...
  public static bool QueueUserWorkItem(WaitCallback callBack);
  public static bool QueueUserWorkItem(WaitCallback callBack,
                                       object state);
}
```

Делегат `WaitCallback` может указывать на любой метод, который принимает в качестве единственного параметра экземпляр `System.Object` (представляющий необязательные данные состояния) и ничего не возвращает. Обратите внимание, что если при вызове `QueueUserWorkItem()` не задается экземпляр `System.Object`, то среда .NET Core Runtime автоматически передает значение `null`. Чтобы продемонстрировать работу методов очередей, работающих с пулом потоков .NET Core Runtime, рассмотрим еще раз программу (в проекте консольного приложения по имени `ThreadPoolApp`), в которой применяется тип `Printer`. На этот раз массив объектов `Thread` не создается вручную, а метод `PrintNumbers()` будет назначаться членам пула потоков:

```
using System;
using System.Threading;
using ThreadPoolApp;

Console.WriteLine("***** Fun with the .NET Core Runtime Thread Pool *****\n");

Console.WriteLine("Main thread started. ThreadID = {0}",
  Thread.CurrentThread.ManagedThreadId);

Printer p = new Printer();

WaitCallback workItem = new WaitCallback(PrintTheNumbers);

// Поставить в очередь метод десять раз.
for (int i = 0; i < 10; i++)
{
  ThreadPool.QueueUserWorkItem(workItem, p);
}
Console.WriteLine("All tasks queued");
Console.ReadLine();

static void PrintTheNumbers(object state)
{
  Printer task = (Printer)state;
  task.PrintNumbers();
}
```

У вас может возникнуть вопрос: почему взаимодействовать с пулом потоков, поддерживаемым средой .NET Core Runtime, выгоднее по сравнению с явным созданием объектов `Thread`? Использование пула потоков обеспечивает следующие преимущества.

- Пул потоков эффективно управляет потоками, сводя к минимуму количество потоков, которые должны создаваться, запускаться и останавливаться.
- За счет применения пула потоков можно сосредоточиться на решении задачи, а не на потоковой инфраструктуре приложения.

Тем не менее, в некоторых случаях ручное управление потоками оказывается более предпочтительным. Ниже приведены примеры.

- Когда требуются потоки переднего плана или должен устанавливаться приоритет потока. Потоки из пула *всегда* являются фоновыми и обладают стандартным приоритетом (`ThreadPriority.Normal`).
- Когда требуется поток с фиксированной идентичностью, чтобы его можно было прерывать, приостанавливать или находить по имени.

На этом исследование пространства имен `System.Threading` завершено. Несомненно, понимание вопросов, рассмотренных в настоящей главе до сих пор (особенно в разделе, посвященном проблемам параллелизма), будет чрезвычайно ценным при создании многопоточного приложения. А теперь, опираясь на имеющийся фундамент, мы переключим внимание на несколько новых аспектов, связанных с потоками, которые появились в .NET 4.0 и остались в .NET Core. Для начала мы обратимся к альтернативной потоковой модели под названием TPL.

Параллельное программирование с использованием TPL

Вы уже ознакомились с объектами из пространства имен `System.Threading`, которые позволяют строить многопоточное программное обеспечение. Начиная с версии .NET 4.0, в Microsoft ввели новый подход к разработке многопоточных приложений, предусматривающий применение библиотеки параллельного программирования, которая называется *TPL*. С помощью типов из `System.Threading.Tasks` можно строить мелкомодульный масштабируемый параллельный код без необходимости напрямую иметь дело с потоками или пулом потоков.

Однако речь не идет о том, что вы не будете использовать типы из пространства имен `System.Threading` во время применения TPL. Оба инструментальных набора для создания многопоточных приложений могут вполне естественно работать вместе. Сказанное особенно верно в связи с тем, что пространство имен `System.Threading` по-прежнему предоставляет большинство примитивов синхронизации, которые рассматривались ранее (`Monitor`, `Interlocked` и т.д.). В итоге вы на самом деле обнаружите, что иметь дело с библиотекой TPL предпочтительнее, чем с первоначальным пространством имен `System.Threading`, т.к. те же самые задачи могут решаться гораздо проще.

Пространство имен System.Threading.Tasks

Все вместе типы из пространства System.Threading.Tasks называются *библиотекой параллельных задач* (Task Parallel Library — TPL). Библиотека TPL будет автоматически распределять нагрузку приложения между доступными процессорами в динамическом режиме с применением пула потоков исполняющей среды. Библиотека TPL поддерживает разбиение работы на части, планирование потоков, управление состоянием и другие низкоуровневые детали. В конечном итоге появляется возможность максимизировать производительность приложений .NET Core, не сталкиваясь со сложностями прямой работы с потоками.

Роль класса Parallel

Основным классом в TPL является System.Threading.Tasks.Parallel. Он содержит методы, которые позволяют осуществлять итерацию по коллекции данных (точнее по объекту, реализующему интерфейс IEnumerable<T>) в параллельной манере. Это делается главным образом посредством двух статических методов Parallel.For() и Parallel.ForEach(), каждый из которых имеет множество перегруженных версий.

Упомянутые методы позволяют создавать тело из операторов кода, которое будет выполняться в параллельном режиме. Концептуально такие операторы представляют логику того же рода, которая была бы написана в нормальной циклической конструкции (с использованием ключевых слов for и foreach языка C#). Преимущество заключается в том, что класс Parallel будет самостоятельно извлекать потоки из пула потоков (и управлять параллелизмом).

Оба метода требуют передачи совместимого с IEnumerable или IEnumerable<T> контейнера, который хранит данные, подлежащие обработке в параллельном режиме. Контейнер может быть простым массивом, необобщенной коллекцией (вроде ArrayList), обобщенной коллекцией (наподобие List<T>) или результатами запроса LINQ.

Вдобавок понадобится применять делегаты System.Func<T> и System.Action<T> для указания целевого метода, который будет вызываться при обработке данных. Делегат Func<T> уже встречался в главе 13 во время исследования технологии LINQ to Objects. Вспомните, что Func<T> представляет метод, который возвращает значение и принимает различное количество аргументов. Делегат Action<T> похож на Func<T> в том, что позволяет задавать метод, принимающий несколько параметров, но данный метод должен возвращать void.

Хотя можно было бы вызывать методы Parallel.For() и Parallel.ForEach() и передавать им строго типизированный объект делегата Func<T> или Action<T>, задача программирования упрощается за счет использования подходящих анонимных методов или лямбда-выражений C#.

Обеспечение параллелизма данных с помощью класса Parallel

Первое применение библиотеки TPL связано с обеспечением *параллелизма данных*. Таким термином обозначается задача прохода по массиву или коллекции в параллельной манере с помощью метода Parallel.For() или Parallel.ForEach(). Предположим, что необходимо выполнить некоторые трудоемкие операции файлового ввода-вывода. В частности, требуется загрузить в память большое число файлов *.jpg, повернуть содержащиеся в них изображения и сохранить модифицированные данные изображений в новом месте.

Задача будет решаться с использованием графического пользовательского интерфейса, так что вы увидите, как применять "анонимные делегаты", позволяющие вторичным потокам обновлять первичный поток пользовательского интерфейса.

На заметку! При построении многопоточного приложения с графическим пользовательским интерфейсом вторичные потоки никогда не смогут напрямую обращаться к элементам управления пользовательского интерфейса. Причина в том, что элементы управления (кнопки, текстовые поля, метки, индикаторы хода работ и т.п.) привязаны к потоку, в котором они создавались. В следующем примере иллюстрируется один из способов обеспечения для вторичных потоков возможности получать доступ к элементам пользовательского интерфейса в безопасной к потокам манере. Во время рассмотрения ключевых слов `async` и `await` языка C# будет предложен более простой подход.

В целях иллюстрации создайте приложение Windows Presentation Foundation (WPF) по имени `DataParallelismWithForEach`, выбрав шаблон WPF App (.NET Core). Чтобы создать проект и добавить его к решению с помощью командной строки, используйте следующие команды:

```
dotnet new wpf -lang c# -n DataParallelismWithForEach
               -o .\DataParallelismWithForEach -f net5.0
dotnet sln .\Chapter15_AllProjects.sln add .\DataParallelismWithForEach
```

На заметку! Инфраструктура Windows Presentation Foundation (WPF) в текущей версии .NET Core предназначена только для Windows и будет подробно рассматриваться в главах 24–28. Если вы еще не работали с WPF, то здесь описано все, что необходимо для данного примера. Разработка приложений WPF ведется в среде Visual Studio Code, хотя никаких визуальных конструкторов там не предусмотрено. Чтобы получить больший опыт разработки приложений WPF, рекомендуется использовать Visual Studio 2019.

Дважды щелкните на имени файла `MainWindow.xaml` в окне Solution Explorer и поместите в него показанное далее содержимое XAML:

```
<Window x:Class="DataParallelismWithForEach.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:DataParallelismWithForEach"
    mc:Ignorable="d"
    Title="Fun with TPL" Height="400" Width="800">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Label Grid.Row="0" Grid.Column="0">
      Feel free to type here while the images are processed...
    </Label>
    <TextBox Grid.Row="1" Grid.Column="0" Margin="10,10,10,10"/>
    <Grid Grid.Row="2" Grid.Column="0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
```

```
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Button Name="cmdCancel" Grid.Row="0" Grid.Column="0"
            Margin="10,10,0,10"
            Click="cmdCancel_Click">
      Cancel
    </Button>
    <Button Name="cmdProcess" Grid.Row="0" Grid.Column="2"
            Margin="0,10,10,10"
            Click="cmdProcess_Click">
      Click to Flip Your Images!
    </Button>
  </Grid>
 </Grid>
</Window>
```

И снова пока не следует задаваться вопросом о том, что означает приведенная разметка или как она работает; вскоре вам придется посвятить немало времени на исследование WPF. Графический пользовательский интерфейс приложения состоит из многострочной текстовой области `TextBox` и одной кнопки `Button` (по имени `cmdProcess`). Текстовая область предназначена для ввода данных во время выполнения работы в фоновом режиме, иллюстрируя тем самым неблокирующую природу параллельной задачи.

В этом примере требуется дополнительный пакет NuGet (`System.Drawing.Common`). Чтобы добавить его в проект, введите следующую команду (целиком в одной строке) в окне командной строки (в каталоге, где находится файл решения) или в консоли диспетчера пакетов в Visual Studio:

```
dotnet add DataParallelismWithForEach package System.Drawing.Common
```

Дважды щелкнув на имени файла `MainWindow.xaml.cs` (может потребоваться развернуть узел `MainWindow.xaml`), добавьте в его начало представленные ниже операторы `using`:

```
// Обеспечить доступ к перечисленным ниже пространствам имен!
// (System.Threading.Tasks уже должно присутствовать благодаря
// выбранному шаблону.)
using System;
using System.Drawing;
using System.Threading.Tasks;
using System.Threading;
using System.Windows;
using System.IO;
```

На заметку! Вы должны обновить строку, передаваемую методу `Directory.GetFiles()`, чтобы в ней был указан конкретный путь к каталогу на вашей машине, который содержит файлы изображений. Для вашего удобства в каталог `TestPictures` включено несколько примеров изображений (поставляемых в составе операционной системы Windows).

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }
  private void cmdCancel_Click(object sender, EventArgs e)
  {
    // Код метода будет вскоре обновлен.
  }
  private void cmdProcess_Click(object sender, EventArgs e)
  {
    ProcessFiles();
    this.Title = "Processing Complete";
  }
  private void ProcessFiles()
  {
    // Загрузить все файлы *.jpg и создать новый каталог
    // для модифицированных данных.
    // Получить путь к каталогу с исполняемым файлом.
    // В режиме отладки VS 2019 текущим каталогом будет
    // <каталог проекта>\bin\debug\net5.0-windows.
    // В случае VS Code или команды dotnet run текущим
    // каталогом будет <каталог проекта>.
    var basePath = Directory.GetCurrentDirectory();
    var pictureDirectory = Path.Combine(basePath, "TestPictures");
    var outputDirectory = Path.Combine(basePath, "ModifiedPictures");
    // Удались любые существующие файлы.
    if (Directory.Exists(outputDirectory))
    {
      Directory.Delete(outputDirectory, true);
    }
    Directory.CreateDirectory(outputDirectory);
    string[] files = Directory.GetFiles(pictureDirectory,
      "*.jpg", SearchOption.AllDirectories);
    // Обработать данные изображений в блокирующей манере.
    foreach (string currentFile in files)
    {
      string filename =
        System.IO.Path.GetFileName(currentFile);
      // Вывести идентификатор потока, обрабатывающего текущее изображение.
      this.Title = $"Processing {filename}
        on thread {Thread.CurrentThread.ManagedThreadId}";
      using (Bitmap bitmap = new Bitmap(currentFile))
      {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(System.IO.Path.Combine(outputDirectory, filename));
      }
    }
  }
}
```

На заметку! В случае получения сообщения об ошибке, связанной с неоднозначностью имени `Path` между `System.IO.Path` и `System.Windows.Shapes.Path`, либо удалите оператор `using` для `System.Windows.Shapes`, либо добавьте `System.IO` к `Path`: `System.IO.Path.Combine(...)`.

Обратите внимание, что метод `ProcessFiles()` выполнит поворот изображения в каждом файле `*.jpg` из указанного каталога. В настоящее время вся работа происходит в первичном потоке исполняемой программы. Следовательно, после щелчка на кнопке Click to Flip Your Images! (Щелкните для поворота ваших изображений) программа выглядит зависшей. Вдобавок заголовок окна также сообщит о том, что файл обрабатывается тем же самым первичным потоком, т.к. в наличии есть только один поток выполнения.

Чтобы обрабатывать файлы на как можно большем количестве процессоров, текущий цикл `foreach` можно заменить вызовом метода `Parallel.ForEach()`. Вспомните, что этот метод имеет множество перегруженных версий. Простейшая форма метода принимает совместимый с `IEnumerable<T>` объект, который содержит элементы, подлежащие обработке (например, строковый массив `files`), и делегат `Action<T>`, указывающий на метод, который будет выполнять необходимую работу.

Ниже приведен модифицированный код, где вместо литерального объекта делегата `Action<T>` применяется лямбда-операция C#. Как видите, в коде *закомментированы* строки, которые отображают идентификатор потока, обрабатывающего текущий файл изображения. Причина объясняется в следующем разделе.

```
// Обработать данные изображений в параллельном режиме!
Parallel.ForEach(files, currentFile =>
  {
    string filename = Path.GetFileName(currentFile);

    // Этот оператор теперь приводит к проблеме! См. следующий раздел.
    // this.Title = $"Processing {filename} on thread
    //     {Thread.CurrentThread.ManagedThreadId}"
    // Thread.CurrentThread.ManagedThreadId);

    using (Bitmap bitmap = new Bitmap(currentFile))
    {
      bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
      bitmap.Save(Path.Combine(outputDirectory, filename));
    }
  }
);
```

Доступ к элементам пользовательского интерфейса во вторичных потоках

Вы наверняка заметили, что в показанном выше коде закомментированы строки, которые обновляют заголовок главного окна значением идентификатора текущего выполняющегося потока. Как упоминалось ранее, элементы управления графического пользовательского интерфейса привязаны к потоку, где они были созданы. Если вторичные потоки пытаются получить доступ к элементу управления, который они напрямую не создавали, то при отладке программного обеспечения возникают ошибки времени выполнения. С другой стороны, если *запустить* приложение (нажатием <Ctrl+F5>), тогда первоначальный код может и не вызвать каких-либо проблем.

На заметку! Не лишним будет повторить: при отладке многопоточного приложения вы иногда будете получать ошибки, когда вторичный поток обращается к элементу управления, созданному в первичном потоке. Однако часто после запуска приложение может выглядеть функционирующим корректно (или же довольно скоро может возникнуть ошибка). Если не предпринять меры предосторожности (описанные далее), то приложение в подобных обстоятельствах может потенциально сгенерировать ошибку во время выполнения.

Один из подходов, который можно использовать для предоставления вторичным потокам доступа к элементам управления в безопасной к потокам манере, предусматривает применение другого приема — *анонимного делегата*. Родительский класс `Control` в WPF определяет объект `Dispatcher`, который управляет рабочими элементами для потока. Указанный объект имеет метод по имени `Invoke()`, принимающий на входе `System.Delegate`. Этот метод можно вызывать внутри кода, выполняющегося во вторичных потоках, чтобы обеспечить возможность безопасного в отношении потоков обновления пользовательского интерфейса для заданного элемента управления. В то время как весь требуемый код делегата можно было бы написать напрямую, большинство разработчиков используют в качестве простой альтернативы синтаксис выражений. Вот как выглядит модифицированный код:

```
// Этот код больше не работает!
// this.Title = $"Processing {filename} on thread {Thread.
CurrentThread.ManagedThreadId}";

// Вызвать Invoke() на объекте Dispatcher, чтобы позволить вторичным потокам
// получать доступ к элементам управления в безопасной к потокам манере.
Dispatcher?.Invoke(() =>
{
  this.Title = $"Processing {filename}";
});
using (Bitmap bitmap = new Bitmap(currentFile))
{
  bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
  bitmap.Save(Path.Combine(outputDirectory, filename));
}
```

Теперь после запуска программы библиотека TPL распределит рабочую нагрузку по множеству потоков из пула, используя столько процессоров, сколько возможно. Тем не менее, поскольку заголовок `Title` всегда обновляется из главного потока, код обновления `Title` больше не отображает текущий поток, и при вводе в текстовой области вы ничего не увидите до тех пор, пока не обработаются все файлы изображений! Причина в том, что первичный поток пользовательского интерфейса по-прежнему блокируется, ожидая завершения работы всех остальных потоков.

Класс `Task`

Класс `Task` позволяет легко вызывать метод во вторичном потоке и может применяться как простая альтернатива асинхронным делегатам. Измените обработчик события `Click` элемента управления `Button` следующим образом:

```
private void cmdProcess_Click(object sender, EventArgs e)
{
  // Запустить новую "задачу" для обработки файлов.
  Task.Factory.StartNew(() => ProcessFiles());
```

```
  // Можно записать и так:
  // Task.Factory.StartNew(ProcessFiles);
}
```

Свойство `Factory` класса `Task` возвращает объект `TaskFactory`. Методу `StartNew()` при вызове передается делегат `Action<T>` (что здесь скрыто с помощью подходящего лямбда-выражения), указывающий на метод, который подлежит вызову в асинхронной манере. После такой небольшой модификации вы обнаружите, что заголовок окна отображает информацию о потоке из пула, обрабатывающем конкретный файл, а текстовое поле может принимать ввод, поскольку пользовательский интерфейс больше не блокируется.

Обработка запроса на отмену

В текущий пример можно внести еще одно улучшение — предоставить пользователю способ для останова обработки данных изображений путем щелчка на второй кнопке Cancel (Отмена). К счастью, методы `Parallel.For()` и `Parallel.ForEach()` поддерживают отмену за счет использования *маркеров отмены*. При вызове методов на объекте `Parallel` им можно передавать объект `ParallelOptions`, который в свою очередь содержит объект `CancellationTokenSource`.

Первым делом определите в производном от `Window` классе закрытую переменную-член `_cancelToken` типа `CancellationTokenSource`:

```
public partial class MainWindow : Window
{
  // Новая переменная уровня Window.
  private CancellationTokenSource _cancelToken =
      new CancellationTokenSource();
  ...
}
```

Обновите обработчик события `Click`:

```
private void cmdCancel_Click(object sender, EventArgs e)
{
  // Используется для сообщения всем рабочим потокам о необходимости останова!
  _cancelToken.Cancel();
}
```

Теперь можно заняться необходимыми модификациями метода `ProcessFiles()`. Вот его финальная реализация:

```
private void ProcessFiles()
{
  // Использовать экземпляр ParallelOptions для хранения CancellationToken.
  ParallelOptions parOpts = new ParallelOptions();
  parOpts.CancellationToken = _cancelToken.Token;
  parOpts.MaxDegreeOfParallelism = System.Environment.ProcessorCount;

  // Загрузить все файлы *.jpg и создать новый каталог
  // для модифицированных данных.
  string[] files = Directory.GetFiles(@".\TestPictures", "*.jpg",
                                      SearchOption. AllDirectories);
  string outputDirectory = @".\ModifiedPictures";
  Directory.CreateDirectory(outputDirectory);
```

```
  try
  {
    // Обработать данные изображения в параллельном режиме!
    Parallel.ForEach(files, parOpts, currentFile =>
    {
      parOpts.CancellationToken.ThrowIfCancellationRequested();
      string filename = Path.GetFileName(currentFile);
      Dispatcher?.Invoke(() =>
      {
        this.Title =
          $"Processing {filename}
            on thread {Thread.CurrentThread.ManagedThreadId}";
      });
      using (Bitmap bitmap = new Bitmap(currentFile))
      {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(outputDirectory, filename));
      }
    });
    Dispatcher?.Invoke(()=>this.Title = "Done!");  // Готово!
  }
  catch (OperationCanceledException ex)
  {
    Dispatcher?.Invoke(()=>this.Title = ex.Message);
  }
}
```

Обратите внимание, что в начале метода конфигурируется объект `Parallel Options` с установкой его свойства `CancellationToken` для применения маркера `CancellationTokenSource`. Кроме того, этот объект `ParallelOptions` передается во втором параметре методу `Parallel.ForEach()`.

Внутри логики цикла осуществляется вызов `ThrowIfCancellationRequested()` на маркере отмены, гарантируя тем самым, что если пользователь щелкнет на кнопке Cancel, то все потоки будут остановлены и в качестве уведомления сгенерируется исключение времени выполнения. Перехватив исключение `OperationCanceledException`, можно добавить в текст главного окна сообщение об ошибке.

Обеспечение параллелизма задач с помощью класса `Parallel`

В дополнение к обеспечению параллелизма данных библиотека TPL также может использоваться для запуска любого количества асинхронных задач с помощью метода `Parallel.Invoke()`. Такой подход немного проще, чем применение делегатов или типов из пространства имен `System.Threading`, но если нужна более высокая степень контроля над выполняемыми задачами, тогда следует отказаться от использования `Parallel.Invoke()` и напрямую работать с классом `Task`, как делалось в предыдущем примере.

Чтобы взглянуть на параллелизм задач в действии, создайте новый проект консольного приложения по имени `MyEBookReader` и импортируйте в начале файла `Program.cs` пространства имен `System.Threading`, `System.Text`, `System.Threading.Tasks`, `System.Linq` и `System.Net` (пример является модификацией полезного примера из документации по .NET Core). Здесь мы будем извлекать публично доступную электронную книгу из сайта проекта Гутенберга (`www.gutenberg.org`) и затем параллельно выполнять набор длительных задач.

Книга загружается в методе GetBook(), показанном ниже:

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Net;
using System.Text;

string _theEBook = "";
GetBook();
Console.WriteLine("Downloading book...");
Console.ReadLine();

void GetBook()
{
  WebClient wc = new WebClient();
  wc.DownloadStringCompleted += (s, eArgs) =>
  {
    _theEBook = eArgs.Result;
    Console.WriteLine("Download complete.");  // Загрузка завершена.
    GetStats();
  };
  // Загрузить электронную книгу Чарльза Диккенса "A Tale of Two Cities".
  // Может понадобиться двукратное выполнение этого кода, если ранее вы
  // не посещали данный сайт, поскольку при первом его посещении появляется
  // окно с сообщением, предотвращающее нормальное выполнение кода.
  wc.DownloadStringAsync(new Uri("http://www.gutenberg.org/
files/98/98-8.txt"));
}
```

Класс WebClient определен в пространстве имен System.Net. Он предоставляет несколько методов для отправки и получения данных от ресурса, идентифицируемого посредством URL. В свою очередь многие из них имеют асинхронные версии, такие как метод DownloadStringAsync(), который автоматически порождает новый поток из пула потоков .NET Core Runtime. Когда объект WebClient завершает получение данных, он инициирует событие DownloadStringCompleted, которое обрабатывается с применением лямбда-выражения C#. Если вызвать синхронную версию этого метода (DownloadString()), то сообщение Downloading book... не появится до тех пор, пока загрузка не завершится.

Далее реализуйте метод GetStats() для извлечения индивидуальных слов, содержащихся в переменной theEBook, и передачи строкового массива на обработку нескольким вспомогательным методам:

```
void GetStats()
{
  // Получить слова из электронной книги.
  string[] words = _theEBook.Split(new char[]
    { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
    StringSplitOptions.RemoveEmptyEntries);

  // Найти 10 наиболее часто встречающихся слов.
  string[] tenMostCommon = FindTenMostCommon(words);

  // Получить самое длинное слово.
  string longestWord = FindLongestWord(words);
```

```
  // Когда все задачи завершены, построить строку, показывающую
  // все статистические данные в окне сообщений.
  StringBuilder bookStats = new StringBuilder("Ten Most Common Words are:\n");
  foreach (string s in tenMostCommon)
  {
    bookStats.AppendLine(s);
  }
  bookStats.AppendFormat("Longest word is: {0}", longestWord);
                        // Самое длинное слово
  bookStats.AppendLine();
  Console.WriteLine(bookStats.ToString(), "Book info");
                                        // Информация о книге
}
```

Метод `FindTenMostCommon()` использует запрос LINQ для получения списка объектов `string`, которые наиболее часто встречаются в массиве `string`, а метод `FindLongestWord()` находит самое длинное слово:

```
string[] FindTenMostCommon(string[] words)
{
  var frequencyOrder = from word in words
                       where word.Length > 6
                       group word by word into g
                       orderby g.Count() descending
                       select g.Key;
  string[] commonWords = (frequencyOrder.Take(10)).ToArray();
  return commonWords;
}
string FindLongestWord(string[] words)
{
  return (from w in words orderby w.Length descending select w)
      .FirstOrDefault();
}
```

После запуска проекта выполнение всех задач может занять внушительный промежуток времени, что зависит от количества процессоров в машине и их тактовой частоты. В конце концов, должен появиться следующий вывод:

```
Downloading book...
Download complete.
Ten Most Common Words are:
Defarge
himself
Manette
through
nothing
business
another
looking
prisoner
Cruncher
Longest word is: undistinguishable
```

Помочь удостовериться в том, что приложение задействует все доступные процессоры машины, может параллельный вызов методов `FindTenMostCommon()` и `FindLongestWord()`. Для этого модифицируйте метод `GetStats()`:

```
void GetStats()
{
  // Получить слова из электронной книги.
  string[] words = _theEBook.Split(
    new char[] { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
    StringSplitOptions.RemoveEmptyEntries);
  string[] tenMostCommon = null;
  string longestWord = string.Empty;

  Parallel.Invoke(
    () =>
    {
      // Найти 10 наиболее часто встречающихся слов.
      tenMostCommon = FindTenMostCommon(words);
    },
    () =>
    {
      // Найти самое длинное слово.
      longestWord = FindLongestWord(words);
    });

  // Когда все задачи завершены, построить строку,
  // показывающую все статистические данные.
  ...
}
```

Метод `Parallel.Invoke()` ожидает передачи в качестве параметра массива делегатов `Action<>`, который предоставляется косвенно с применением лямбда-выражения. В то время как вывод идентичен, преимущество заключается в том, что библиотека TPL теперь будет использовать все доступные процессоры машины для вызова каждого метода параллельно, если подобное возможно.

Запросы Parallel LINQ (PLINQ)

В завершение знакомства с библиотекой TPL следует отметить, что существует еще один способ встраивания параллельных задач в приложения .NET Core. При желании можно применять набор расширяющих методов, которые позволяют конструировать запрос LINQ, распределяющий свою рабочую нагрузку по параллельным потокам (когда это возможно). Соответственно запросы LINQ, которые спроектированы для параллельного выполнения, называются *запросами Parallel LINQ* (*PLINQ*).

Подобно параллельному коду, написанному с использованием класса `Parallel`, в PLINQ имеется опция игнорирования запроса на обработку коллекции параллельным образом, если понадобится. Инфраструктура PLINQ оптимизирована во многих отношениях, включая определение того, не будет ли запрос на самом деле более эффективно выполняться в синхронной манере.

Во время выполнения PLINQ анализирует общую структуру запроса, и если есть вероятность, что запрос выиграет от распараллеливания, то он будет выполняться параллельно. Однако если распараллеливание запроса ухудшит производительность, то PLINQ просто запустит запрос последовательно. Когда возникает выбор между

потенциально затратным (в плане ресурсов) параллельным алгоритмом и экономным последовательным, предпочтение по умолчанию отдается последовательному алгоритму.

Необходимые расширяющие методы находятся в классе `ParallelEnumerable` из пространства имен `System.Linq`. В табл. 15.5 описаны некоторые полезные расширяющие методы PLINQ.

Таблица 15.5. Избранные члены класса `ParallelEnumerable`

Член	Назначение
`AsParallel()`	Указывает, что остаток запроса должен быть по возможности распараллелен
`WithCancellation()`	Указывает, что инфраструктура PLINQ должна периодически отслеживать состояние предоставленного маркера отмены и при необходимости отменять выполнение
`WithDegreeOfParallelism()`	Указывает максимальное количество процессоров, которое инфраструктура PLINQ должна задействовать при распараллеливании запроса
`ForAll()`	Позволяет обрабатывать результаты параллельно без предварительного слияния с потоком потребителя, как происходит при перечислении результата LINQ с применением ключевого слова `foreach`

Чтобы взглянуть на PLINQ в действии, создайте проект консольного приложения по имени `PLINQDataProcessingWithCancellation` и импортируйте в него пространства имен `System.Linq`, `System.Threading` и `System.Threading.Tasks` (если это еще не сделано). После начала обработки запускается новая задача, выполняющая запрос LINQ, который просматривает крупный массив целых чисел в поиске элементов, удовлетворяющих условию, что остаток от их деления на 3 дает 0. Вот *непараллельная* версия такого запроса:

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

Console.WriteLine("Press any key to start processing");
                  // Нажмите любую клавишу, чтобы начать обработку
Console.ReadKey();

Console.WriteLine("Processing");
Task.Factory.StartNew(ProcessIntData);
Console.ReadLine();

void ProcessIntData()
{
  // Получить очень большой массив целых чисел.
  int[] source = Enumerable.Range(1, 10_000_000).ToArray();
  // Найти числа, для которых истинно условие num % 3 == 0,
  // и возвратить их в убывающем порядке.
  int[] modThreeIsZero = (
    from num in source
```

```
    where num % 3 == 0
    orderby num descending
    select num).ToArray();
  // Вывести количество найденных чисел.
  Console.WriteLine($"Found {modThreeIsZero.Count()} numbers that
match query!");
}
```

Создание запроса PLINQ

Чтобы проинформировать библиотеку TPL о выполнении запроса в параллельном режиме (если такое возможно), необходимо использовать расширяющий метод `AsParallel()`:

```
int[] modThreeIsZero = (
  from num in source
  where num % 3 == 0
  orderby num descending select num).ToArray();
```

Обратите внимание, что общий формат запроса LINQ идентичен тому, что вы видели в предыдущих главах. Тем не менее, за счет включения вызова `AsParallel()` библиотека TPL попытается распределить рабочую нагрузку по доступным процессорам.

Отмена запроса PLINQ

С помощью объекта `CancellationTokenSource` запрос PLINQ можно также информировать о прекращении обработки при определенных условиях (обычно из-за вмешательства пользователя). Объявите на уровне класса `Program` объект `CancellationTokenSource` по имени `_cancelToken` и модифицируйте операторы верхнего уровня для принятия ввода от пользователя. Ниже показаны соответствующие изменения в коде:

```
CancellationTokenSource _cancelToken =
  new CancellationTokenSource();

do
{
  Console.WriteLine("Press any key to start processing");
                  // Нажмите любую клавишу, чтобы начать обработку
  Console.ReadKey();
  Console.WriteLine("Processing");
  Task.Factory.StartNew(ProcessIntData);
  Console.Write("Enter Q to quit: ");
              // Введите Q для выхода:
  string answer = Console.ReadLine();

  // Желает ли пользователь выйти?
  if (answer.Equals("Q", StringComparison.OrdinalIgnoreCase))
  {
    _cancelToken.Cancel();
    break;
  }
}
while (true);

Console.ReadLine();
```

Теперь запрос PLINQ необходимо информировать о том, что он должен ожидать входящего запроса на отмену выполнения, добавив в цепочку вызов расширяющего метода `WithCancellation()` с передачей ему маркера отмены. Кроме того, этот запрос PLINQ понадобится поместить в подходящий блок `try/catch` и обработать возможные исключения. Финальная версия метода `ProcessInData()` выглядит следующим образом:

```
static void ProcessIntData()
{
  // Получить очень большой массив целых чисел.
  int[] source = Enumerable.Range(1, 10_000_000).ToArray();

  // Найти числа, для которых истинно условие num % 3 == 0,
  // и возвратить их в убывающем порядке.
  int[] modThreeIsZero = null;

  try
  {
    modThreeIsZero =
(from num in source.AsParallel().WithCancellation(_cancelToken.Token)
                   where num % 3 == 0
                   orderby num descending
                   select num).ToArray();
    Console.WriteLine();

    // Вывести количество найденных чисел.
    Console.WriteLine($"Found {modThreeIsZero.Count()} numbers that
match query!");
  }
  catch (OperationCanceledException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Во время выполнения метода `ProcessIntData()` понадобится нажать <Q> и быстро произвести ввод, чтобы увидеть сообщение от маркера отмены.

Асинхронные вызовы с помощью `async/await`

В этой довольно длинной главе было представлено много материала в сжатом виде. Конечно, построение, отладка и понимание сложных многопоточных приложений требует прикладывания усилий в любой инфраструктуре. Хотя TPL, PLINQ и тип делегата могут до некоторой степени упростить решение (особенно по сравнению с другими платформами и языками), разработчики по-прежнему должны хорошо знать детали разнообразных расширенных приемов.

С выходом версии .NET 4.5 в языке программирования C# появились два новых ключевых слова, которые дополнительно упрощают процесс написания асинхронного кода. По контрасту со всеми примерами, показанными ранее в главе, когда применяются ключевые слова `async` и `await`, компилятор будет самостоятельно генерировать большой объем кода, связанного с потоками, с использованием многочисленных членов из пространств имен `System.Threading` и `System.Threading.Tasks`.

Знакомство с ключевыми словами async и await языка C# (обновление в версиях 7.1, 9.0)

Ключевое слово async языка C# применяется для указания на то, что метод, лямбда-выражение или анонимный метод должен вызываться в асинхронной манере *автоматически*. Да, это правда. Благодаря простой пометке метода модификатором async среда .NET Core Runtime будет создавать новый поток выполнения для обработки текущей задачи. Более того, при вызове метода async ключевое слово await будет *автоматически* приостанавливать текущий поток до тех пор, пока задача не завершится, давая возможность вызывающему потоку продолжить свою работу.

В целях иллюстрации создайте новый проект консольного приложения по имени FunWithCSharpAsync и импортируйте в файл Program.cs пространства имен System.Threading, System.Threading.Task и System.Collections.Generic. Добавьте метод DoWork(), который заставляет вызывающий поток ожидать пять секунд. Ниже показан код:

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

Console.WriteLine("Fun With Async ===>");
Console.WriteLine(DoWork());
Console.WriteLine("Completed");
Console.ReadLine();

static string DoWork()
{
  Thread.Sleep(5_000);
  return "Done with work!";
}
```

Вам известно, что после запуска программы придется ожидать пять секунд, прежде чем сможет произойти что-то еще. В случае графического приложения весь пользовательский интерфейс был бы заблокирован до тех пор, пока работа не завершится.

Если бы мы решили прибегнуть к одному из описанных ранее приемов, чтобы сделать приложение более отзывчивым, тогда пришлось бы немало потрудиться. Тем не менее, начиная с версии .NET 4.5, можно написать следующий код C#:

```
...
string message = await DoWorkAsync();
Console.WriteLine(message);
...
static string DoWork()
{
  Thread.Sleep(5_000);
  return "Done with work!";
}
static async Task<string> DoWorkAsync()
{
  return await Task.Run(() =>
  {
    Thread.Sleep(5_000);
    return "Done with work!";
  });
}
```

Если вы используете в качестве точки входа метод `Main()` (вместо операторов верхнего уровня), тогда должны пометить метод с помощью ключевого слова `async`, появившегося в версии C# 7.1:

```
static async Task Main(string[] args)
{
  ...
  string message = await DoWorkAsync();
  Console.WriteLine(message);
  ...
}
```

На заметку! Возможность декорирования метода `Main()` посредством `async` — нововведение, появившееся в версии C# 7.1. Операторы верхнего уровня в версии C# 9.0 являются неявно асинхронными.

Обратите внимание на ключевое слово `await` *перед* именем метода, который будет вызван в неблокирующей манере. Это важно: если метод декорируется ключевым словом `async`, но не имеет хотя бы одного внутреннего вызова метода с использованием `await`, то получится синхронный вызов (на самом деле компилятор выдаст соответствующее предупреждение).

Кроме того, вы должны применять класс `Task` из пространства имен `System.Threading.Tasks` для переделки методов `Main()` (если вы используете `Main()`) и `DoWork()` (последний добавляется как `DoWorkAsync()`). По существу вместо возвращения просто специфического значения (объекта `string` в текущем примере) возвращается объект `Task<T>`, где обобщенный параметр типа `T` представляет собой действительное возвращаемое значение.

Реализация метода `DoWorkAsync()` теперь напрямую возвращает объект `Task<T>`, который является возвращаемым значением `Task.Run()`. Метод `Run()` принимает делегат `Func<>` или `Action<>` и, как вам уже известно, для простоты здесь можно использовать лямбда-выражение. В целом новая версия `DoWorkAsync()` может быть описана следующим образом.

При вызове запускается новая задача, которая заставляет вызывающий поток уснуть на пять секунд. После завершения вызывающий поток предоставляет строковое возвращаемое значение. Эта строка помещается в новый объект `Task<string>` и возвращается вызывающему коду.

Благодаря новой реализации метода `DoWorkAsync()` мы можем получить некоторое представление о подлинной роли ключевого слова `await`. Оно всегда будет модифицировать метод, который возвращает объект `Task`. Когда поток выполнения достигает `await`, вызывающий поток приостанавливается до тех пор, пока вызов не будет завершен. Запустив эту версию приложения, вы обнаружите, что сообщение `Completed` отображается перед сообщением `Done with work!`. В случае графического приложения можно было бы продолжать работу с пользовательским интерфейсом одновременно с выполнением метода `DoWorkAsync()`.

Класс `SynchronizationContext` и `async/await`

Тип `SynchronizationContext` формально определен как базовый класс, который предоставляет свободный от потоков контекст баз синхронизации. Хотя такое первоначальное определение не особо информативно, в официальной документации указаны следующие сведения.

Цель модели синхронизации, реализуемой классом `SynchronizationContext`*, заключается в том, чтобы позволить внутренним асинхронным/синхронным операциям общеязыковой исполняющей среды вести себя надлежащим образом с различными моделями синхронизации.*

Наряду с тем, что вам уже известно о многопоточности, такое заявление проливает свет на этот вопрос. Вспомните, что приложения с графическим пользовательским интерфейсом (Windows Forms, WPF) не разрешают прямой доступ к элементам управления из вторичных потоков, а требуют делегирования доступа. Вы уже видели объект `Dispatcher` в примере приложения WPF. В консольных приложениях, которые не используют WPF, это ограничение отсутствует. Речь идет о разных моделях синхронизации. С учетом всего сказанного давайте рассмотрим класс `SynchronizationContext`.

Класс `SynchonizationContext` является типом, предоставляющим виртуальный метод отправки, который принимает делегат, предназначенный для выполнения асинхронным образом. В результате инфраструктуры получают шаблон для надлежащей обработки асинхронных запросов (диспетчеризация для приложений WPF/Windows Forms, прямое выполнение для приложений без графического пользовательского интерфейса и т.д.). Он предлагает способ постановки в очередь единицы работы в контексте и подсчета асинхронных операций, ожидающих выполнения.

Как обсуждалось ранее, когда делегат помещается в очередь для асинхронного выполнения, он планируется к запуску в отдельном потоке, что обрабатывается средой .NET Core Runtime. Задача обычно решается с помощью управляемого пула потоков .NET Core Runtime, но может быть построена и специальная реализация.

Хотя такими связующими действиями можно управлять вручную в коде, шаблон `async/await` делает большую часть трудной работы. В случае применения `await` к асинхронному методу задействуются реализации `SynchronizationContext` и `TaskScheduler` целевой инфраструктуры. Например, если вы используете `async/await` в приложении WPF, то инфраструктура WPF обеспечит диспетчеризацию делегата и обратный вызов в конечном автомате при завершении ожидающей задачи, чтобы безопасным образом обновить элементы управления.

Роль метода `ConfigureAwait()`

Теперь, когда вы лучше понимаете роль класса `SynchronizationContext`, пришло время раскрыть роль метода `ConfigureAwait()`. По умолчанию применение `await` к объекту `Task` приводит к использованию контекста синхронизации. При разработке приложений с графическим пользовательским интерфейсом (Windows Forms, WPF) именно такое поведение является желательным. Однако в случае написания кода приложения без графического пользовательского интерфейса накладные расходы, связанные с постановкой в очередь исходного контекста, когда в этом нет нужды, потенциально могут вызвать проблемы с производительностью приложения.

Чтобы увидеть все в действии, модифицируйте операторы верхнего уровня, как показано ниже:

```
Console.WriteLine(" Fun With Async ===>");
// Console.WriteLine(DoWork());
string message = await DoWorkAsync();
Console.WriteLine(message);

string message1 = await DoWorkAsync().ConfigureAwait(false);
Console.WriteLine(message1);
```

В исходном блоке кода применяется класс SynchronizationContext, поставляемый инфраструктурой (в данном случае средой .NET Core Runtime), что эквивалентно вызову ConfigureAwait(true). Во втором примере текущий контекст и планировщик игнорируются.

Согласно рекомендациям команды создателей .NET Core при разработке прикладного кода (Windows Forms, WPF и т.д.) следует полагаться на стандартное поведение, а в случае написания неприкладного кода (скажем, библиотеки) использовать вызов ConfigureAwait(false). Одним исключением является инфраструктура ASP.NET Core (рассматриваемая в части IX), где специальная реализация SynchronizationContext не создается; таким образом, вызов ConfigureAwait(false) не дает преимущества при работе с другими инфраструктурами.

Соглашения об именовании асинхронных методов

Конечно же, вы заметили, что мы изменили имя метода с DoWork() на DoWorkAsync(), но по какой причине? Давайте предположим, что новая версия метода по-прежнему называется DoWork(), но вызывающий код реализован так:

```
// Отсутствует ключевое слово await!
string message = DoWork();
```

Обратите внимание, что мы действительно пометили метод ключевым словом async, но не указали ключевое слово await при вызове DoWork(). Здесь мы получим ошибки на этапе компиляции, потому что возвращаемым значением DoWork() является объект Task, который мы пытаемся напрямую присвоить переменной типа string. Вспомните, что ключевое слово await отвечает за извлечение внутреннего возвращаемого значения, которое содержится в объекте Task. Поскольку await отсутствует, возникает несоответствие типов.

На заметку! Метод, поддерживающий await — это просто метод, который возвращает Task или Task<T>.

С учетом того, что методы, которые возвращают объекты Task, теперь могут вызываться в неблокирующей манере посредством конструкций async и await, в Microsoft рекомендуют (в качестве установившейся практики) снабжать имя любого метода, возвращающего Task, суффиксом Async. В таком случае разработчики, которым известно данное соглашение об именовании, получают визуальное напоминание о том, что ключевое слово await является обязательным, если они намерены вызывать метод внутри асинхронного контекста.

На заметку! Обработчики событий для элементов управления графического пользовательского интерфейса (вроде обработчика события Click кнопки), а также методы действий внутри приложений в стиле MVC, к которым применяются ключевые слова async и await, не следуют указанному соглашению об именовании.

Асинхронные методы, возвращающие void

В настоящий момент наш метод DoWorkAsync() возвращает объект Task, содержащий "реальные данные" для вызывающего кода, которые будут получены прозрачным образом через ключевое слово await. Однако что если требуется построить асинхронный метод, возвращающий void? Реализация зависит о того, нуждается метод в применении await или нет (как в сценариях "запустил и забыл").

Асинхронные методы, возвращающие `void` и поддерживающие `await`

Если асинхронный метод должен поддерживать `await`, тогда используйте необобщенный класс `Task` и опустите любые операторы `return`, например:

```
static async Task MethodReturningTaskOfVoidAsync()
{
  await Task.Run(() => { /* Выполнить какую-то работу... */
                         Thread.Sleep(4_000);
                       });
  Console.WriteLine("Void method completed");  // Метод завершен
}
```

Затем в коде, вызывающем этот метод, примените ключевое слово `await`:

```
await MethodReturningVoidAsync();
Console.WriteLine("Void method complete");
```

Асинхронные методы, возвращающие `void` и работающие в стиле "запустил и забыл"

Если метод должен быть асинхронным, но не обязан поддерживать `await` и применяться в сценариях "запустил и забыл", тогда добавьте ключевое слово `async` и сделайте возвращаемым типом `void`, а не `Task`. Методы такого рода обычно используются для задач вроде ведения журнала, когда нежелательно, чтобы запись в журнал приводила к задержке выполнения остального кода.

```
static async void MethodReturningVoidAsync()
{
  await Task.Run(() => { /* Выполнить какую-то работу... */
                         Thread.Sleep(4_000);
                       });
  Console.WriteLine("Fire and forget void method completed");
                  // Метод завершен
}
```

Затем в коде, вызывающем этот метод, ключевое слово `await` *не* используется:

```
MethodReturningVoidAsync();
Console.WriteLine("Void method complete");
```

Асинхронные методы с множеством контекстов `await`

Внутри реализации асинхронного метода разрешено иметь множество контекстов `await`. Следующий код является вполне допустимым:

```
static async Task MultipleAwaits()
{
  await Task.Run(() => { Thread.Sleep(2_000); });
  Console.WriteLine("Done with first task!");
                  // Первая задача завершена!

  await Task.Run(() => { Thread.Sleep(2_000); });
  Console.WriteLine("Done with second task!");
                  // Вторая задача завершена!

  await Task.Run(() => { Thread.Sleep(2_000); });
  Console.WriteLine("Done with third task!"); //Третья задача завершена!
}
```

Здесь каждая задача всего лишь приостанавливает текущий поток на некоторый период времени; тем не менее, посредством таких задач может быть представлена любая единица работы (обращение к веб-службе, чтение базы данных или что-нибудь еще).

Еще один вариант предусматривает ожидание не каждой отдельной задачи, а всех их вместе. Это более вероятный сценарий, когда имеются три работы (скажем, проверка поступления сообщений электронной почты, обновление сервера, загрузка файлов), которые должны делаться в пакете, но могут выполняться параллельно. Ниже приведен модифицированный код, в котором используется метод `Task.WhenAll()`:

```
static async Task MultipleAwaits()
{
  var task1 = Task.Run(() =>
  {
    Thread.Sleep(2_000);
    Console.WriteLine("Done with first task!");
  });
  var task2=Task.Run(() =>
  {
    Thread.Sleep(1_000);
    Console.WriteLine("Done with second task!");
  });
  var task3 = Task.Run(() =>
  {
    Thread.Sleep(1_000);
    Console.WriteLine("Done with third task!");
  });
  await Task.WhenAll(task1, task2, task3);
}
```

Запустив программу, вы увидите, что три задачи запускаются в порядке от наименьшего значения, указанного при вызове метода `Sleep()`:

```
Fun With Async ===>
Done with work!
Void method completed
Done with second task!
Done with third task!
Done with first task!
Completed
```

Существует также метод `WhenAny()`, возвращающий задачу, которая завершилась. Для демонстрации работы `WhenAny()` измените последнюю строку метода `MultipleAwaits()` следующим образом:

```
await Task.WhenAny(task1, task2, task3);
```

В результате вывод становится таким:

```
Fun With Async ===>
Done with work!
Void method completed
Done with second task!
Completed
Done with third task!
Done with first task!
```

Вызов асинхронных методов из неасинхронных методов

В каждом из предшествующих примеров ключевое слово `async` использовалось для возвращения в поток вызывающего кода, пока выполняется асинхронный метод. В целом ключевое слово `await` может применяться только в методе, помеченном как `async`. А что если вы не можете (или не хотите) помечать метод с помощью `async`?

К счастью, существуют другие способы вызова асинхронных методов. Если вы просто не используете ключевое слово `await`, тогда код продолжает работу после асинхронного метода, не возвращая управление вызывающему коду. Если вам необходимо ожидать завершения асинхронного метода (что происходит, когда применяется ключевое слово `await`), то существуют два подхода.

Первый подход предусматривает просто использование свойства `Result` с методами, возвращающими `Task<T>`, или метода `Wait()` с методами, возвращающими `Task`/`Task<T>`. (Вспомните, что метод, который возвращает значение, обязан возвращать `Task<T>`, будучи асинхронным, а метод, не имеющий возвращаемого значения, возвращает `Task`, когда является асинхронным.) Если метод терпит неудачу, то возвращается `AggregateException`.

Можете также добавить вызов `GetAwaiter().GetResult()`, который обеспечивает такой же эффект, как ключевое слово `await` в асинхронном методе, и распространяет исключения в той же манере, что и `async`/`await`. Тем не менее, указанные методы помечены в документации как "не предназначенные для внешнего использования", а это значит, что они могут измениться либо вовсе исчезнуть в какой-то момент в будущем. Вызов `GetAwaiter().GetResult()` работает как с методами, возвращающими значение, так и с методами без возвращаемого значения.

На заметку! Решение использовать свойство `Result` или вызов `GetAwaiter().GetResult()` с `Task<T>` возлагается полностью на вас, и большинство разработчиков принимают решение, основываясь на обработке исключений. Если ваш метод возвращает `Task`, тогда вы должны применять вызов `GetAwaiter().GetResult()` или `Wait()`.

Например, вот как вы могли бы вызывать метод `DoWorkAsync()`:

```
Console.WriteLine(DoWorkAsync().Result);
Console.WriteLine(DoWorkAsync().GetAwaiter().GetResult());
```

Чтобы остановить выполнение до тех пор, пока не произойдет возврат из метода с возвращаемым типом `void`, просто вызовите метод `Wait()` на объекте `Task`:

```
MethodReturningVoidAsync().Wait();
```

Ожидание с помощью `await` в блоках `catch` и `finally`

В версии C# 6 появилась возможность помещения вызовов `await` в блоки `catch` и `finally`. Для этого сам метод обязан быть `async`. Указанная возможность демонстрируется в следующем примере кода:

```
static async Task<string> MethodWithTryCatch()
{
  try
  {
    // Выполнить некоторую работу.
    return "Hello";
  }
```

```
  catch (Exception ex)
  {
    await LogTheErrors();
    throw;
  }
  finally
  {
    await DoMagicCleanUp();
  }
}
```

Обобщенные возвращаемые типы в асинхронных методах (нововведение в версии 7.0)

До выхода версии C# 7 возвращаемыми типами методов `async` были только `Task`, `Task<T>` и `void`. В версии C# 7 доступны дополнительные возвращаемые типы при условии, что они следуют шаблону с ключевым словом `async`. В качестве конкретного примера можно назвать тип `ValueTask`. Введите код, подобный показанному ниже:

```
static async ValueTask<int> ReturnAnInt()
{
  await Task.Delay(1_000);
  return 5;
}
```

К типу `ValueTask` применимы все те же самые правила, что и к типу `Task`, поскольку `ValueTask` — это просто объект `Task` для типов значений, заменяющий собой принудительное размещение объекта в куче.

Локальные функции (нововведение в версии 7.0)

Локальные функции были представлены в главе 4 и использовались в главе 8 с итераторами. Они также могут оказаться полезными для асинхронных методов. Чтобы продемонстрировать преимущество, сначала нужно взглянуть на проблему. Добавьте новый метод по имени `MethodWithProblems()` со следующим кодом:

```
static async Task MethodWithProblems(int firstParam, int secondParam)
{
  Console.WriteLine("Enter");
  await Task.Run(() =>
  {
    // Вызвать длительно выполняющийся метод.
    Thread.Sleep(4_000);
    Console.WriteLine("First Complete");

    // Вызвать еще один длительно выполняющийся метод, который терпит
    // неудачу из-за того, что значение второго параметра выходит
    // за пределы допустимого диапазона.
    Console.WriteLine("Something bad happened");
  });
}
```

Сценарий заключается в том, что вторая длительно выполняющаяся задача терпит неудачу из-за недопустимых входных данных. Вы можете (и должны) добавить в начало метода проверки, но поскольку весь метод является асинхронным, нет ни-

каких гарантий, что такие проверки выполнятся. Было бы лучше, чтобы проверки происходили непосредственно перед выполнением вызываемого кода. В приведенном далее обновленном коде проверки делаются в синхронной манере, после чего закрытая функция выполняется асинхронным образом.

```
static async Task MethodWithProblemsFixed(int firstParam,
                                          int secondParam)
{
  Console.WriteLine("Enter");
  if (secondParam < 0)
  {
    Console.WriteLine("Bad data");
    return;
  }
  await actualImplementation();
  async Task actualImplementation()
  {
    await Task.Run(() =>
    {
      // Вызвать длительно выполняющийся метод.
      Thread.Sleep(4_000);
      Console.WriteLine("First Complete");
      // Вызвать еще один длительно выполняющийся метод, который терпит
      // неудачу из-за того, что значение второго параметра выходит
      // за пределы допустимого диапазона.
      Console.WriteLine("Something bad happened");
                        // Что-то пошло не так.
    });
  }
}
```

Отмена операций `async/await`

Шаблон `async/await` также допускает отмену, которая реализуется намного проще, чем с методом `Parallel.ForEach()`. Для демонстрации будет применяться тот же самый проект приложения WPF, рассмотренный ранее в главе. Вы можете либо повторно использовать этот проект, либо создать в решении новый проект приложения WPF (.NET Core) и добавить к нему пакет `System.Drawing.Common` с помощью следующих команд CLI:

```
dotnet new wpf -lang c# -n PictureHandlerWithAsyncAwait
               -o .\PictureHandlerWithAsyncAwait -f net5.0
dotnet sln .\Chapter15_AllProjects.sln add
           .\PictureHandlerWithAsyncAwait
dotnet add PictureHandlerWithAsyncAwait package System.Drawing.Common
```

Если вы работаете в Visual Studio, тогда щелкните правой кнопкой мыши на имени решения в окне Solution Explorer, выберите в контекстном меню пункт Add⇨Project (Добавить⇨Проект) и назначьте ему имя `PictureHandlerWithAsyncAwait`. Сделайте новый проект стартовым, щелкнув правой кнопкой мыши на его имени и выбрав в контекстном меню пункт Set as StartUp Project (Установить как стартовый проект). Добавьте NuGet-пакет `System.Drawing.Common`:

```
dotnet add PictureHandlerWithAsyncAwait package System.Drawing.Common
```

Приведите разметку XAML в соответствие с предыдущим проектом приложения WPF, но с заголовком `Picture Handler with Async/Await`.

Удостоверьтесь, что в файле `MainWindow.xaml.cs` присутствуют показанные ниже операторы `using`:

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Drawing;
```

Затем добавьте переменную уровня класса для объекта `CancellationToken` и обработчик событий для кнопки Cancel:

```
private CancellationTokenSource _cancelToken = null;
private void cmdCancel_Click(object sender, EventArgs e)
{
  _cancelToken.Cancel();
}
```

Процесс здесь такой же, как в предыдущем примере: получение каталога с файлами изображений, создание выходного каталога, получение файлов, поворот изображений в файлах и сохранение их в выходном каталоге. В новой версии для выполнения работы будут применяться асинхронные методы, а не `Parallel.ForEach()`, и сигнатуры методов принимают в качестве параметра объект `CancellationToken`. Введите следующий код:

```
private async void cmdProcess_Click(object sender, EventArgs e)
{
  _cancelToken = new CancellationTokenSource();
  var basePath = Directory.GetCurrentDirectory();
  var pictureDirectory =
    Path.Combine(basePath, "TestPictures");
  var outputDirectory =
    Path.Combine(basePath, "ModifiedPictures");
  // Удалить любые существующие файлы.
  if (Directory.Exists(outputDirectory))
  {
    Directory.Delete(outputDirectory, true);
  }
  Directory.CreateDirectory(outputDirectory);
  string[] files = Directory.GetFiles(
    pictureDirectory, "*.jpg", SearchOption.AllDirectories);
  try
  {
    foreach(string file in files)
    {
      try
      {
        await ProcessFile(
          file, outputDirectory,_cancelToken.Token);
      }
```

```
      catch (OperationCanceledException ex)
      {
        Console.WriteLine(ex);
        throw;
      }
    }
  }
  catch (OperationCanceledException ex)
  {
    Console.WriteLine(ex);
    throw;
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
    throw;
  }
  _cancelToken = null;
  this.Title = "Processing complete";  // Обработка завершена.
}
```

После начальных настроек в коде организуется цикл по файлам с асинхронным вызовом метода `ProcessFile()` для каждого файла. Вызов метода `ProcessFile()` помещен внутрь блока `try/catch` и ему передается объект `CancellationToken`. Если вызов `Cancel()` выполняется на `CancellationTokenSource` (т.е. когда пользователь щелкает на кнопке Cancel), тогда генерируется исключение `OperationCanceledException`.

На заметку! Код `try/catch` может находиться где угодно в цепочке вызовов (как вскоре вы увидите). Размещать его при первом вызове или внутри самого асинхронного метода — вопрос личных предпочтений и нужд приложения.

Наконец, добавьте финальный метод `ProcessFile()`:

```
private async Task ProcessFile(string currentFile,
    string outputDirectory, CancellationToken token)
{
  string filename = Path.GetFileName(currentFile);
  using (Bitmap bitmap = new Bitmap(currentFile))
  {
    try
    {
      await Task.Run(() =>
      {
        Dispatcher?.Invoke(() =>
        {
          this.Title = $"Processing {filename}";
        });
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(outputDirectory, filename));
      }
      ,token);
    }
```

```
    catch (OperationCanceledException ex)
    {
      Console.WriteLine(ex);
      throw;
    }
  }
}
```

Метод `ProcessFile()` использует еще одну перегруженную версию `Task.Run()`, которая принимает в качестве параметра объект `CancellationToken`. Вызов `Task.Run()` помещен внутрь блока `try/catch` (как и вызывающий код) на случай щелчка пользователем на кнопке Cancel.

Асинхронные потоки (нововведение в версии 8.0)

В версии C# 8.0 появилась возможность создания и потребления потоков данных (раскрываются в главе 20) асинхронным образом. Метод, который возвращает асинхронный поток данных:

- объявляется с модификатором `async`;
- возвращает реализацию `IAsyncEnumerable<T>`;
- содержит операторы `yield return` (рассматривались в главе 8) для возвращения последовательных элементов в асинхронном потоке данных.

Взгляните на приведенный далее пример:

```
public static async IAsyncEnumerable<int> GenerateSequence()
{
  for (int i = 0; i < 20; i++)
  {
    await Task.Delay(100);
    yield return i;
  }
}
```

Метод `GenerateSequence()` объявлен как `async`, возвращает реализацию `IAsyncEnumerable<int>` и применяет `yield return` для возвращения целых чисел из последовательности. Чтобы вызывать этот метод, добавьте следующий код:

```
await foreach (var number in GenerateSequence())
{
  Console.WriteLine(number);
}
```

Итоговые сведения о ключевых словах `async` и `await`

Настоящий раздел содержал много примеров; ниже перечислены ключевые моменты, которые в нем рассматривались.

- Методы (а также лямбда-выражения или анонимные методы) могут быть помечены ключевым словом `async`, что позволяет им работать в неблокирующей манере.
- Методы (а также лямбда-выражения или анонимные методы), помеченные ключевым словом `async`, будут выполняться синхронно до тех пор, пока не встретится ключевое слово `await`.

- Один метод `async` может иметь множество контекстов `await`.
- Когда встречается выражение `await`, вызывающий поток приостанавливается до тех пор, пока ожидаемая задача не завершится. Тем временем управление возвращается коду, вызвавшему метод.
- Ключевое слово `await` будет скрывать с глаз возвращаемый объект `Task`, что выглядит как прямой возврат лежащего в основе возвращаемого значения. Методы, не имеющие возвращаемого значения, просто возвращают `void`.
- Проверка параметров и другая обработка ошибок должна делаться в главной части метода с переносом фактической порции `async` в закрытую функцию.
- Для переменных, находящихся в стеке, объект `ValueTask` более эффективен, чем объект `Task`, который может стать причиной упаковки и распаковки.
- По соглашению об именовании методы, которые могут вызываться асинхронно, должны быть помечены с помощью суффикса `Async`.

Резюме

Глава начиналась с исследования роли пространства имен `System.Threading`. Как было показано, когда приложение создает дополнительные потоки выполнения, в результате появляется возможность выполнять множество задач (по виду) одновременно. Также было продемонстрировано несколько способов защиты чувствительных к потокам блоков кода, чтобы предотвратить повреждение разделяемых ресурсов.

Затем в главе исследовались новые модели для разработки многопоточных приложений, введенные в .NET 4.0, в частности Task Parallel Library и PLINQ. В завершение главы была раскрыта роль ключевых слов `async` и `await`. Вы видели, что эти ключевые слова используются многими типами в библиотеке TPL; однако большинство работ по созданию сложного кода для многопоточной обработки и синхронизации компилятор выполняет самостоятельно.

ЧАСТЬ V

Программирование с использованием сборок .NET Core

ГЛАВА 16

Построение и конфигурирование библиотек классов

В большинстве примеров, рассмотренных до сих пор, создавались "автономные" исполняемые приложения, где вся программная логика упаковывалась в единственную сборку (`*.dll`) и выполнялась с применением `dotnet.exe` (или копии `dotnet.exe`, носящей имя сборки). Такие сборки использовали в основном библиотеки базовых классов .NET Core. В то время как некоторые простые программы .NET Core могут быть сконструированы с применением только библиотек базовых классов, многократно используемая программная логика нередко изолируется в *специальных* библиотеках классов (файлах `*.dll`), которые могут разделяться между приложениями.

В настоящей главе вы сначала исследуете детали разнесения типов по пространствам имен .NET Core. После этого вы подробно ознакомитесь с библиотеками классов в .NET Core, выясните разницу между .NET Core и .NET Standard, а также научитесь конфигурировать приложения, публиковать консольные приложения .NET Core и упаковывать свои библиотеки в многократно используемые пакеты NuGet.

Определение специальных пространств имен

Прежде чем погружаться в детали развертывания и конфигурирования библиотек, сначала необходимо узнать, каким образом упаковывать свои специальные типы в пространства имен .NET Core. Вплоть до этого места в книге создавались небольшие тестовые программы, которые задействовали существующие пространства имен из мира .NET Core (в частности `System`). Однако когда строится крупное приложение со многими типами, возможно, будет удобно группировать связанные типы в специальные пространства имен. В C# такая цель достигается с применением ключевого слова `namespace`. Явное определение специальных пространств имен становится еще более важным при построении разделяемых сборок, т.к. для использования ваших типов другие разработчики будут нуждаться в ссылке на вашу библиотеку и импортировании специальных пространств имен. Специальные пространства имен также предотвращают конфликты имен, отделяя ваши специальные классы от других специальных классов, которые могут иметь совпадающие имена.

Чтобы исследовать все аспекты непосредственно, начните с создания нового проекта консольного приложения .NET Core под названием `CustomNamespaces`.

Предположим, что требуется разработать коллекцию геометрических классов с именами Square (квадрат), Circle (круг) и Hexagon (шестиугольник). Учитывая сходные между ними черты, было бы желательно сгруппировать их в уникальном пространстве имен MyShapes внутри сборки CustomNamespaces.exe.

Хотя компилятор C# без проблем воспримет единственный файл кода C#, содержащий множество типов, такой подход может стать проблематичным в командном окружении. Если вы работаете над типом Circle, а ваш коллега — над типом Hexagon, тогда вам придется по очереди работать с монолитным файлом или сталкиваться с трудноразрешимыми (во всяком случае, отнимающими много времени) конфликтами при слиянии изменений.

Более удачный подход предусматривает помещение каждого класса в собственный файл, с определением в каждом из них пространства имен. Чтобы обеспечить упаковку типов в ту же самую логическую группу, просто помещайте заданные определения классов в область действия одного и того же пространства имен:

```
// Circle.cs
namespace MyShapes
{
  // Класс Circle.
  public class Circle { /* Интересные методы... */ }
}
// Hexagon.cs
namespace MyShapes
{
  // Класс Hexagon.
  public class Hexagon { /* Еще интересные методы... */ }
}
// Square.cs
namespace MyShapes
{
  // Класс Square.
  public class Square { /* И еще интересные методы... */ }
}
```

На заметку! Рекомендуется иметь в каждом файле кода только один класс. В ряде приводимых ранее примеров такое правило не соблюдалось, но причиной было упрощение изучения. В последующих главах каждый класс по возможности будет располагаться в собственном файле кода.

Обратите внимание на то, что пространство MyShapes действует как концептуальный "контейнер" для определяемых в нем классов. Когда в другом пространстве имен (скажем, CustomNamespaces) необходимо работать с типами из отдельного пространства имен, вы применяете ключевое слово using, как поступали бы в случае использования пространств имен из библиотек базовых классов .NET Core:

```
// Обратиться к пространству имен из библиотек базовых классов.
using System;

// Использовать типы, определенные в пространстве имен MyShapes.
using MyShapes;
Hexagon h = new Hexagon();
Circle c = new Circle();
Square s = new Square();
```

В примере предполагается, что файлы C#, где определено пространство имен MyShapes, являются частью того же самого проекта консольного приложения; другими словами, все эти файлы компилируются в единственную сборку. Если пространство имен MyShapes определено во внешней сборке, то для успешной компиляции потребуется также добавить ссылку на данную библиотеку. На протяжении настоящей главы вы изучите все детали построения приложений, взаимодействующих с внешними библиотеками.

Разрешение конфликтов имен с помощью полностью заданных имен

Говоря формально, вы не обязаны применять ключевое слово using языка C# при ссылках на типы, определенные во внешних пространствах имен. Вы можете использовать *полностью заданные имена* типов, которые, как упоминалось в главе 1, представляют собой имена типов, предваренные названиями пространств имен, где типы определены. Например:

```
// Обратите внимание, что пространство имен MyShapes больше
// не импортируется!
using System;
MyShapes.Hexagon h = new MyShapes.Hexagon();
MyShapes.Circle c = new MyShapes.Circle();
MyShapes.Square s = new MyShapes.Square();
```

Обычно необходимость в применении полностью заданных имен отсутствует. Они требуют большего объема клавиатурного ввода, но никак не влияют на размер кода и скорость выполнения. На самом деле в коде CIL типы *всегда* определяются с полностью заданными именами. С этой точки зрения ключевое слово using языка C# является просто средством экономии времени на наборе.

Тем не менее, полностью заданные имена могут быть полезными (а иногда и необходимыми) для избегания потенциальных конфликтов имен при использовании множества пространств имен, которые содержат идентично названные типы. Предположим, что есть новое пространство имен My3DShapes, где определены три класса, которые способны визуализировать фигуры в трехмерном формате:

```
// Еще одно пространство имен для работы с фигурами.
// Circle.cs
namespace My3DShapes
{
  // Класс для представления трехмерного круга.
  public class Circle { }
}
// Hexagon.cs
namespace My3DShapes
{
  // Класс для представления трехмерного шестиугольника.
  public class Hexagon { }
}
// Square.cs
namespace My3DShapes
{
  // Класс для представления трехмерного квадрата.
  public class Square { }
}
```

Если теперь вы модифицируете операторы верхнего уровня, как показано ниже, то получите несколько ошибок на этапе компиляции, потому что в обоих пространствах имен определены одинаково именованные классы:

```
// Масса неоднозначностей!
using System;
using MyShapes;
using My3DShapes;

// На какое пространство имен производится ссылка?
Hexagon h = new Hexagon();  // Ошибка на этапе компиляции!
Circle c = new Circle();    // Ошибка на этапе компиляции!
Square s = new Square();    // Ошибка на этапе компиляции!
```

Устранить неоднозначности можно за счет применения полностью заданных имен:

```
// Теперь неоднозначности устранены.
My3DShapes.Hexagon h = new My3DShapes.Hexagon();
My3DShapes.Circle c = new My3DShapes.Circle();
MyShapes.Square s = new MyShapes.Square();
```

Разрешение конфликтов имен с помощью псевдонимов

Ключевое слово `using` языка C# также позволяет создавать псевдоним для полностью заданного имени типа. В этом случае определяется метка, которая на этапе компиляции заменяется полностью заданным именем типа. Определение псевдонимов предоставляет второй способ разрешения конфликтов имен. Вот пример:

```
using System;
using MyShapes;
using My3DShapes;

// Устранить неоднозначность, используя специальный псевдоним.
using The3DHexagon = My3DShapes.Hexagon;

// На самом деле здесь создается экземпляр класса My3DShapes.Hexagon.
The3DHexagon h2 = new The3DHexagon();
...
```

Продемонстрированный альтернативный синтаксис `using` также дает возможность создавать псевдонимы для пространств имен с очень длинными названиями. Одним из пространств имен с самым длинным названием в библиотеках базовых классов является `System.Runtime.Serialization.Formatters.Binary`, которое содержит член по имени `BinaryFormatter`. При желании экземпляр класса `BinaryFormatter` можно создать следующим образом:

```
using bfHome = System.Runtime.Serialization.Formatters.Binary;
bfHome.BinaryFormatter b = new bfHome.BinaryFormatter();
...
```

либо с использованием традиционной директивы `using`:

```
using System.Runtime.Serialization.Formatters.Binary;
BinaryFormatter b = new BinaryFormatter();
...
```

На данном этапе не нужно беспокоиться о предназначении класса `BinaryFormatter` (он исследуется в главе 20). Сейчас просто запомните, что ключевое слово `using` в

C# позволяет создавать псевдонимы для очень длинных полностью заданных имен или, как случается более часто, для разрешения конфликтов имен, которые могут возникать при импорте пространств имен, определяющих типы с идентичными названиями.

На заметку! Имейте в виду, что чрезмерное применение псевдонимов C# в результате может привести к получению запутанной кодовой базы. Если другие программисты в команде не знают о ваших специальных псевдонимах, то они могут полагать, что псевдонимы ссылаются на типы из библиотек базовых классов, и прийти в замешательство, не обнаружив их описания в документации.

Создание вложенных пространств имен

При организации типов допускается определять пространства имен внутри других пространств имен. В библиотеках базовых классов подобное встречается во многих местах и обеспечивает размещение типов на более глубоких уровнях. Например, пространство имен `IO` вложено внутрь пространства имен `System`, давая в итоге `System.IO`.

Шаблоны проектов .NET Core помещают начальный код в файле `Program.cs` внутрь пространства имен, название которого совпадает с именем проекта. Такое базовое пространство имен называется *корневым*. В этом примере корневым пространством имен, созданным шаблоном .NET Core, является `CustomNamespaces`:

```
namespace CustomNamespaces
{
  class Program
  {
    ...
  }
}
```

На заметку! В случае замены комбинации `Program/Main()` операторами верхнего уровня назначить им какое-либо пространство имен не удастся.

Вложить пространства имен `MyShapes` и `My3DShapes` внутрь корневого пространства имен можно двумя способами. Первый — просто вложить ключевое слово `namespace`, например:

```
namespace CustomNamespaces
{
  namespace MyShapes
  {
    // Класс Circle.
    public class Circle
    {
      /* Интересные методы... */
    }
  }
}
```

Второй (и более распространенный) способ предусматривает использование "точечной записи" в определении пространства имен, как показано ниже:

```
namespace CustomNamespaces.MyShapes
{
  // Класс Circle.
  public class Circle
  {
    /* Интересные методы... */
  }
}
```

Пространства имен не обязаны содержать какие-то типы непосредственно, что позволяет применять их для обеспечения дополнительного уровня области действия.

Учитывая, что теперь пространство `My3DShapes` вложено внутрь корневого пространства имен `CustomNamespaces`, вам придется обновить все существующие директивы `using` и псевдонимы типов (при условии, что вы модифицировали все примеры классов с целью их вложения внутрь корневого пространства имен):

```
using The3DHexagon = CustomNamespaces.My3DShapes.Hexagon;
using CustomNamespaces.MyShapes;
```

На заметку! На практике принято группировать файлы в пространстве имен по каталогам. Вообще говоря, расположение файла в рамках структуры каталогов никак не влияет на пространства имен. Однако такой подход делает структуру пространств имен более ясной (и конкретной) для других разработчиков. По этой причине многие разработчики и инструменты анализа кода ожидают соответствия пространств имен структуре каталогов.

Изменение стандартного пространства имен в Visual Studio

Как упоминалось ранее, при создании нового проекта C# с использованием Visual Studio (либо интерфейса .NET Core CLI) название корневого пространства имен приложения будет совпадать с именем проекта. Когда затем в Visual Studio к проекту добавляются новые файлы кода с применением пункта меню Project⇨Add New Item (Проект⇨Добавить новый элемент), типы будут автоматически помещаться внутрь корневого пространства имен. Если вы хотите изменить название корневого пространства имен, тогда откройте окно свойств проекта, перейдите в нем на вкладку Application (Приложение) и введите желаемое имя в поле Default namespace (Стандартное пространство имен), как показано на рис. 16.1.

На заметку! В окне свойств проекта Visual Studio корневое пространство имен по-прежнему представлено как стандартное (default). Далее вы увидите, почему в книге оно называется корневым (root) пространством имен.

Конфигурировать корневое пространство имен можно также путем редактирования файла проекта (`*.csproj`). Чтобы открыть файл проекта .NET Core, дважды щелкните на его имени в окне Solution Explorer или щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт Edit project file (Редактировать файл проекта). После открытия файла обновите главный узел `PropertyGroup`, добавив узел `RootNamespace`:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <RootNamespace>CustomNamespaces2</RootNamespace>
  </PropertyGroup>
</Project>
```

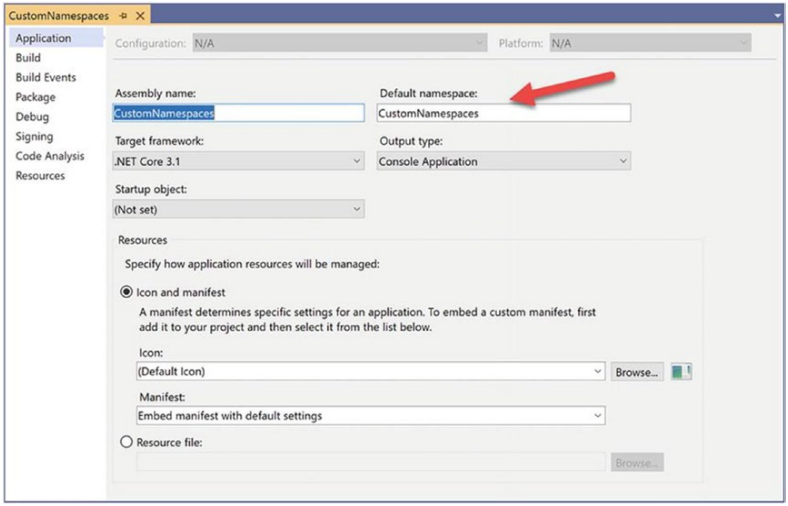

Рис. 16.1. Конфигурирование стандартного пространства имен

Теперь, когда вы ознакомились с некоторыми деталями упаковки специальных типов в четко организованные пространства имен, давайте кратко рассмотрим преимущества и формат сборки .NET Core. Затем мы углубимся в подробности создания, развертывания и конфигурирования специальных библиотек классов.

Роль сборок .NET Core

Приложения .NET Core конструируются путем соединения в одно целое любого количества *сборок*. Выражаясь просто, сборка представляет собой самоописательный двоичный файл, который поддерживает версии и обслуживается средой .NET Core Runtime. Невзирая на то, что сборки .NET Core имеют такие же файловые расширения (`*.exe` или `*.dll`), как и старые двоичные файлы Windows, в их внутренностях мало общего. Таким образом, первым делом давайте выясним, какие преимущества предлагает формат сборки.

Сборки содействуют многократному использованию кода

При построении проектов консольных приложений в предшествующих главах могло показаться, что *вся* функциональность приложений содержалась внутри конструируемых исполняемых сборок. В действительности примеры приложений задействовали многочисленные типы из всегда доступных библиотек базовых классов.NET Core.

Возможно, вы уже знаете, что *библиотека кода* (также называемая *библиотекой классов*) — это файл `*.dll`, который содержит типы, предназначенные для применения внешними приложениями. При построении исполняемых сборок вы без сомнения будете использовать много системных и специальных библиотек кода по мере создания приложений. Однако имейте в виду, что библиотека кода необязательно должна получать файловое расширение `*.dll`. Вполне допускается (хотя нечасто), чтобы исполняемая сборка работала с типами, определенными внутри внешнего исполняемого файла. В таком случае ссылаемый файл `*.exe` также может считаться библиотекой кода.

Независимо от того, как упакована библиотека кода, платформа .NET Core позволяет многократно применять типы в независимой от языка манере. Например, вы могли бы создать библиотеку кода на C# и повторно использовать ее при написании кода на другом языке программирования .NET Core. Между языками есть возможность не только выделять память под экземпляры типов, но также и наследовать от самих типов. Базовый класс, определенный в C#, может быть расширен классом, написанным на Visual Basic. Интерфейсы, определенные в F#, могут быть реализованы структурами, определенными в C#, и т.д. Важно понимать, что за счет разбиения единственного монолитного исполняемого файла на несколько сборок .NET Core достигается возможность многократного использования кода в форме, *нейтральной к языку*.

Сборки устанавливают границы типов

Вспомните, что *полностью заданное имя* типа получается за счет предварения имени этого типа (`Console`) названием пространства имен, где он определен (`System`). Тем не менее, выражаясь строго, удостоверение типа дополнительно устанавливается сборкой, в которой он находится. Например, если есть две уникально именованных сборки (`MyCars.dll` и `YourCars.dll`), которые определяют пространство имен (`CarLibrary`), содержащее класс по имени `SportsCar`, то в мире .NET Core такие типы `SportsCar` будут рассматриваться как уникальные.

Сборки являются единицами, поддерживающими версии

Сборкам .NET Core назначается состоящий из четырех частей числовой номер версии в форме *<старший номер>.<младший номер>.<номер сборки>.<номер редакции>*. (Если номер версии явно не указан, то сборке автоматически назначается версия 1.0.0.0 из-за стандартных настроек проекта в .NET Core.) Этот номер позволяет множеству версий той же самой сборки свободно сосуществовать на одной машине.

Сборки являются самоописательными

Сборки считаются *самоописательными* отчасти из-за того, что в своем *манифесте* содержат информацию обо всех внешних сборках, к которым они должны иметь доступ для корректного функционирования. Вспомните из главы 1, что манифест представляет собой блок метаданных, которые описывают саму сборку (имя, версия, обязательные внешние сборки и т.д.).

В дополнение к данным манифеста сборка содержит метаданные, которые описывают структуру каждого содержащегося в ней типа (имена членов, реализуемые интерфейсы, базовые классы, конструкторы и т.п.). Благодаря тому, что сборка настолько детально документирована, среда .NET Core Runtime *не* нуждается в обращении к реестру Windows для выяснения ее местонахождения (что радикально отличается от унаследованной модели программирования COM от Microsoft). Такое отделение от реестра является одним из факторов, которые позволяют приложениям .NET Core функционировать под управлением других операционных систем (ОС) помимо Windows, а также обеспечивают поддержку на одной машине множества версий платформы .NET Core.

В текущей главе вы узнаете, что для получения информации о местонахождении внешних библиотек кода среда .NET Core Runtime применяет совершенно новую схему.

Формат сборки .NET Core

Теперь, когда вы узнали о многих преимуществах сборок .NET Core, давайте более детально рассмотрим, как такие сборки устроены внутри. Говоря о структуре, сборка .NET Core (`*.dll` или `*.exe`) состоит из следующих элементов:

- заголовок файла ОС (например, Windows);
- заголовок файла CLR;
- код CIL;
- метаданные типов;
- манифест сборки;
- дополнительные встроенные ресурсы.

Несмотря на то что первые два элемента (заголовки ОС и CLR) представляют собой блоки данных, которые обычно можно игнорировать, краткого рассмотрения они все же заслуживают. Ниже приведен обзор всех перечисленных элементов.

Установка инструментов профилирования C++

В последующих нескольких разделах используется утилита по имени `dumpbin.exe`, которая входит в состав инструментов профилирования C++. Чтобы установить их, введите *C++ profiling* в поле быстрого поиска и щелкните на подсказке Install C++ profiling tools (Установить инструменты профилирования C++), как показано на рис. 16.2.

Рис. 16.2. Установка инструментов профилирования C++ в Visual Studio

В результате запустится установщик Visual Studio с выбранными инструментами. В качестве альтернативы можете запустить установщик Visual Studio самостоятельно и выбрать необходимые компоненты (рис. 16.3).

Заголовок файла операционной системы (Windows)

Заголовок файла ОС устанавливает факт того, что сборку можно загружать и манипулировать ею в среде целевой ОС (Windows в рассматриваемом примере). Данные в этом заголовке также идентифицируют вид приложения (консольное, с графическим пользовательским интерфейсом или библиотека кода `*.dll`), которое должно обслуживаться ОС.

Откройте файл `CarLibrary.dll` (в хранилище GitHub для книги или созданный позже в главе) с применением утилиты `dumpbin.exe` (в окне командной строки разработчика), указав ей флаг `/headers`:

```
dumpbin /headers CarLibrary.dll
```

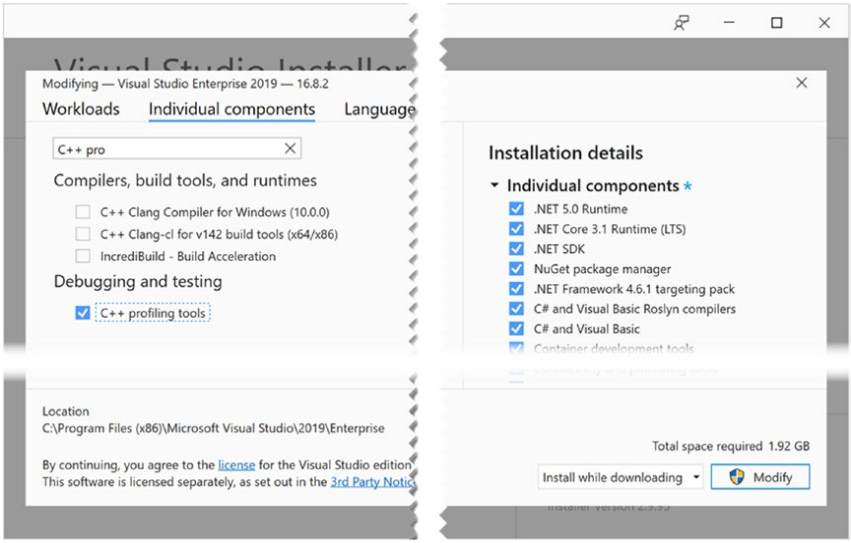

Рис. 16.3. Альтернативная установка инструментов профилирования C++

В результате отобразится информация заголовка файла ОС сборки, построенной для Windows, часть которой показана ниже:

```
Dump of file carlibrary.dll
PE signature found
File Type: DLL

FILE HEADER VALUES
      14C machine (x86)
        3 number of sections
   BB89DC3D time date stamp
        0 file pointer to symbol table
        0 number of symbols
       E0 size of optional header
     2022 characteristics
            Executable
            Application can handle large (>2GB) addresses
            DLL
...
Дамп файла CarLibrary.dll

Обнаружена подпись PE
Тип файла: DLL

Значения заголовка файла
      14C машина (x86)
        3 количество разделов
   BB89DC3D дата и время
        0 файловый указатель на таблицу символов
        0 количество символов
       E0 размер необязательного заголовка
```

```
        2022 характеристики
             Исполняемый файл
             Приложение может обрабатывать большие (> 2 Гбайт) адреса
             DLL
...
```

Запомните, что подавляющему большинству программистов, использующих .NET Core, никогда не придется беспокоиться о формате данных заголовков, встроенных в сборку .NET Core. Если только вы не занимаетесь разработкой нового компилятора языка .NET Core (в таком случае вы *обязаны* позаботиться о подобной информации), то можете не вникать в тонкие детали заголовков. Однако помните, что такая информация потребляется "за кулисами", когда ОС загружает двоичный образ в память.

Заголовок файла CLR

Заголовок файла CLR — это блок данных, который должны поддерживать все сборки .NET Core (и благодаря компилятору C# они его поддерживают), чтобы обслуживаться средой .NET Core Runtime. Выражаясь кратко, в заголовке CLR определены многочисленные флаги, которые позволяют исполняющей среде воспринимать компоновку управляемого файла. Например, существуют флаги, идентифицирующие местоположение метаданных и ресурсов внутри файла, версию исполняющей среды, для которой была построена сборка, значение (необязательного) открытого ключа и т.д. Снова запустите утилиту `dumpbin.exe`, указав флаг `/clrheader`:

```
dumpbin /clrheader CarLibrary.dll
```

Вы увидите внутреннюю информацию заголовка файла CLR для заданной сборки .NET Core:

```
Dump of file CarLibrary.dll
File Type: DLL

  clr Header:

    48 cb
  2.05 runtime version
  2158 [ B7C] RVA [size] of MetaData Directory
     1 flags
         IL Only
     0 entry point token
     0 [   0] RVA [size] of Resources Directory
     0 [   0] RVA [size] of StrongNameSignature Directory
     0 [   0] RVA [size] of CodeManagerTable Directory
     0 [   0] RVA [size] of VTableFixups Directory
     0 [   0] RVA [size] of ExportAddressTableJumps Directory
     0 [   0] RVA [size] of ManagedNativeHeader Directory

  Summary

        2000 .reloc
        2000 .rsrc
        2000 .text
```

```
Дамп файла CarLibrary.dll
Тип файла: DLL

  Заголовок clr:
    48 cb
  2.05 версия исполняющей среды
```

```
2164 [    A74] RVA [размер] каталога MetaData
  1 флаги
     Только IL
  0 маркер записи
  0 [   0] RVA [размер] каталога Resources
  0 [   0] RVA [размер] каталога StrongNameSignature
  0 [   0] RVA [размер] каталога CodeManagerTable
  0 [   0] RVA [размер] каталога VTableFixups
  0 [   0] RVA [размер] каталога ExportAddressTableJumps
  0 [   0] RVA [размер] каталога ManagedNativeHeader

Сводка
     2000 .reloc
     2000 .rsrc
     2000 .text
```

И снова важно отметить, что вам как разработчику приложений .NET Core не придется беспокоиться о тонких деталях информации заголовка файла CLR. Просто знайте, что каждая сборка .NET Core содержит данные такого рода, которые исполняющая среда .NET Core использует "за кулисами" при загрузке образа в память. Теперь переключите свое внимание на информацию, которая является намного более полезной при решении повседневных задач программирования.

Код CIL, метаданные типов и манифест сборки

В своей основе сборка содержит код CIL, который, как вы помните, представляет собой промежуточный язык, не зависящий от платформы и процессора. Во время выполнения внутренний код CIL на лету посредством JIT-компилятора компилируется в инструкции, специфичные для конкретной платформы и процессора. Благодаря такому проектному решению сборки .NET Core действительно могут выполняться под управлением разнообразных архитектур, устройств и ОС. (Хотя вы можете благополучно и продуктивно работать, не разбираясь в деталях языка программирования CIL, в главе 19 предлагается введение в синтаксис и семантику CIL.)

Сборка также содержит метаданные, полностью описывающие формат внутренних типов и формат внешних типов, на которые сборка ссылается. Исполняющая среда .NET Core применяет эти метаданные для выяснения местоположения типов (и их членов) внутри двоичного файла, для размещения типов в памяти и для упрощения удаленного вызова методов. Более подробно детали формата метаданных .NET Core будут раскрыты в главе 17 во время исследования служб рефлексии.

Сборка должна также содержать связанный с ней *манифест* (по-другому называемый *метаданными сборки*). Манифест документирует каждый *модуль* внутри сборки, устанавливает версию сборки и указывает любые *внешние* сборки, на которые ссылается текущая сборка. Как вы увидите далее в главе, исполняющая среда .NET Core интенсивно использует манифест сборки в процессе нахождения ссылок на внешние сборки.

Дополнительные ресурсы сборки

Наконец, сборка .NET Core может содержать любое количество встроенных ресурсов, таких как значки приложения, файлы изображений, звуковые клипы или таблицы строк. На самом деле платформа .NET Core поддерживает *подчиненные сборки*, которые содержат только локализованные ресурсы и ничего другого. Они могут быть удобны, когда необходимо отделять ресурсы на основе культуры (русской, немецкой,

английской и т.д.) при построении интернационального программного обеспечения. Тема создания подчиненных сборок выходит за рамки настоящей книги; если вам интересно, обращайтесь за информацией о подчиненных сборках и локализации в документацию по .NET Core.

Отличия между библиотеками классов и консольными приложениями

До сих пор в этой книге почти все примеры были консольными приложениями .NET Core. При наличии опыта разработки для .NET, вы заметите, что они похожи на консольные приложения .NET. Основное отличие касается процесса конфигурирования (рассматривается позже), а также того, что они выполняются под управлением .NET Core. Консольные приложения имеют единственную точку входа (либо указанный метод `Main()`, либо операторы верхнего уровня), способны взаимодействовать с консолью и могут запускаться прямо из среды ОС. Еще одно отличие между консольными приложениями .NET Core и .NET связано с тем, что консольные приложения в .NET Core запускаются с применением хоста приложений .NET Core (`dotnet.exe`).

С другой стороны, библиотеки классов не имеют точки входа и потому не могут запускаться напрямую. Они используются для инкапсуляции логики, специальных типов и т.п., а ссылка на них производится из других библиотек классов и/или консольных приложений. Другими словами, библиотеки классов применяются для хранения всего того, о чем шла речь в разделе "Роль сборок .NET Core" ранее в главе.

Отличия между библиотеками классов .NET Standard и .NET Core

Библиотеки классов .NET Core функционируют под управлением .NET Core, а библиотеки классов .NET — под управлением .NET. Все довольно просто. Тем не менее, здесь имеется проблема. Предположим, что ваша организация располагает крупной кодовой базой .NET, разрабатываемой в течение (потенциально) многих лет вами и коллегами по команде. Возможно, существует совместно используемый код значительного объема, задействованный в приложениях, которые вы и ваша команда создали за прошедшие годы. Вполне вероятно, что этот код реализует централизованное ведение журнала, формирование отчетов или функциональность, специфичную для предметной области.

Теперь вы (вместе с вашей организацией) хотите вести разработку новых приложений с применением .NET Core. А что делать со всем совместно используемым кодом? Переписывание унаследованного кода для его помещения в сборки .NET Core может требовать значительных усилий. Вдобавок до тех пор, пока все ваши приложения не будут перенесены в .NET Core, вам придется поддерживать две версии (одну в .NET и одну в .NET Core), что приведет к резкому снижению продуктивности.

К счастью, разработчики платформы .NET Core продумали такой сценарий. В .NET Core появился .NET Standard — новый тип проекта библиотеки классов, на которую можно ссылаться в приложениях как .NET, так и .NET Core. Однако прежде чем выяснять, оправданы ли ваши ожидания, следует упомянуть об одной загвоздке с .NET (Core) 5, которая будет рассмотрена чуть позже.

В каждой версии .NET Standard определен общий набор API-интерфейсов, которые должны поддерживаться всеми версиями .NET (.NET, .NET Core, Xamarin и т.д.),

чтобы удовлетворять требованиям стандарта. Например, если бы вы строили библиотеку классов как проект .NET Standard 2.0, то на нее можно было бы ссылаться из .NET 4.6.1+ и .NET Core 2.0+ (плюс разнообразные версии Xamarin, Mono, Universal Windows Platform и Unity).

Это означает, что вы могли бы перенести код из своих библиотек классов .NET в библиотеки классов .NET Standard 2.0 и совместно использовать их в приложениях .NET Core и .NET. Такое решение гораздо лучше, чем поддержка двух копий того же самого кода, по одной для каждой платформы.

А теперь собственно о загвоздке. Каждая версия .NET Standard представляет собой наименьший общий знаменатель для платформ, которые она поддерживает, т.е. чем ниже версия, тем меньше вы можете делать в своей библиотеке классов.

Хотя в .NET (Core) 5 и .NET Core 3.1 можно ссылаться на библиотеку .NET Standard 2.0, в такой библиотеке вам не удастся задействовать существенное количество функциональных средств C# 8.0 (или любых средств C# 9.0). Для полной поддержки C# 8.0 и C# 9.0 вы должны применять .NET Standard 2.1, а .NET Standard 2.0 подходит только для .NET 4.8 (самая поздняя/последняя версия первоначальной инфраструктуры .NET Framework).

Итак, .NET Standard — все еще хороший механизм для использования существующего кода в более новых приложениях, но он не является панацеей.

Конфигурирование приложений

В то время как всю информацию, необходимую вашему приложению .NET Core, допускается хранить в исходном коде, наличие возможности изменять определенные значения во время выполнения жизненно важно в большинстве приложений. Обычно это делается посредством конфигурационного файла, который поставляется вместе с приложением.

На заметку! В предшествующих версиях .NET Framework конфигурация приложений базировалась на файле XML по имени `app.config` (или `web.config` для приложений ASP.NET). Хотя конфигурационные XML-файлы по-прежнему можно применять, как будет показано в текущем разделе, главный способ конфигурирования приложений .NET Core предусматривает использование файлов JSON (JavaScript Object Notation — запись объектов JavaScript). Конфигурация будет подробно обсуждаться в главах, посвященных WPF и ASP.NET Core.

Чтобы ознакомиться с процессом, создайте новый проект консольного приложения .NET Core по имени `FunWithConfiguration` и добавьте к нему ссылку на пакет `Microsoft.Extensions.Configuration.Json`:

```
dotnet new console -lang c# -n FunWithConfiguration
                   -o .\FunWithConfiguration -f net5.0
dotnet add FunWithConfiguration
  package Microsoft.Extensions.Configuration.Json
```

Команды добавят к вашему проекту ссылку на подсистему конфигурации .NET Core, основанную на файлах JSON (вместе с необходимыми зависимостями). Чтобы задействовать ее, добавьте в проект новый файл JSON по имени `appsettings.json`. Модифицируйте файл проекта, обеспечив копирование этого файла в выходной каталог при каждой компиляции проекта:

```
<ItemGroup>
  <None Update="appsettings.json">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

Приведите содержимое файла `appsettings.json` к следующему виду:

```
{
  "CarName": "Suzy"
}
```

На заметку! Если вы не знакомы с форматом JSON, то знайте, что он представляет собой формат с парами "имя-значение" и объектами, заключенными в фигурные скобки. Целый файл может быть прочитан как один объект, а подобъекты тоже помечаются с помощью фигурных скобок. Позже в книге вы будете иметь дело с более сложными файлами JSON.

Финальный шаг связан с чтением конфигурационного файла и получением значения `CarName`. Обновите операторы `using` в файле `Program.cs`, как показано ниже:

```
using System;
using System.IO;
using Microsoft.Extensions.Configuration;
```

Модифицируйте метод `Main()` следующим образом:

```
static void Main(string[] args)
{
  IConfiguration config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json", true, true)
    .Build();
}
```

Новая подсистема конфигурации начинается с создания экземпляра класса `ConfigurationBuilder`. Он позволяет добавлять множество файлов, устанавливать свойства (такие как местоположение конфигурационных файлов) и, в конце концов, встраивать конфигурацию внутрь экземпляра реализации интерфейса `IConfiguration`.

Имея экземпляр реализации `IConfiguration`, вы можете обращаться с ним так, как принято в версии .NET 4.8. Добавьте приведенный далее код в конец метода `Main()` и после запуска приложения вы увидите, что значение будет выведено на консоль:

```
Console.WriteLine($"My car's name is {config["CarName"]}");
Console.ReadLine();
```

В дополнение к файлам JSON существуют пакеты для поддержки переменных среды, Azure Key Vault, аргументов командной строки и многого другого. Подробные сведения ищите в документации по .NET Core.

Построение и потребление библиотеки классов .NET Core

Чтобы заняться исследованием мира библиотек классов .NET Core, будет создана сборка `*.dll` (по имени `CarLibrary`), содержащая небольшой набор открытых типов. Для начала создайте решение. Затем создайте проект библиотеки классов по имени `CarLibrary` и добавьте его в решение, если это еще не делалось.

```
dotnet new sln -n Chapter16_AllProjects
dotnet new classlib -lang c# -n CarLibrary -o .\CarLibrary -f net5.0
dotnet sln .\Chapter16_AllProjects.sln add .\CarLibrary
```

Первая команда создает в текущем каталоге пустой файл решения по имени `Chapter16_AllProjects` (`-n`). Вторая команда создает новый проект библиотеки классов .NET 5.0 (`-f`) под названием `CarLibrary` (`-n`) в подкаталоге `CarLibrary` (`-o`). Указывать выходной подкаталог (`-o`) необязательно. Если он опущен, то проект будет создан в подкаталоге с таким же именем, как у проекта. Третья команда добавляет новый проект к решению.

На заметку! Интерфейс командной строки .NET Core снабжен хорошей справочной системой. Для получения сведений о любой команде укажите с ней `-h`. Например, чтобы увидеть все шаблоны, введите `dotnet new -h`. Для получения дополнительной информации о создании проекта библиотеки классов введите `dotnet new classlib -h`.

После создания проекта и решения вы можете открыть его в Visual Studio (или Visual Studio Code), чтобы приступить к построению классов. Открыв решение, удалите автоматически сгенерированный файл `Class1.cs`. Проектное решение библиотеки для работы с автомобилями начинается с создания перечислений `EngineStateEnum` и `MusicMediaEnum`. Добавьте в проект два файла с именами `MusicMediaEnum.cs` и `EngineStateEnum.cs` и поместите в них следующий код:

```
// MusicMediaEnum.cs
namespace CarLibrary
{
  // Тип музыкального проигрывателя, установленный в данном автомобиле.
  public enum MusicMediaEnum
  {
    MusicCd,
    MusicTape,
    MusicRadio,
    MusicMp3
  }
}
// EngineStateEnum.cs
namespace CarLibrary
{
  // Представляет состояние двигателя.
  public enum EngineStateEnum
  {
    EngineAlive,
    EngineDead
  }
}
```

Далее создайте абстрактный базовый класс по имени `Car`, который определяет разнообразные данные состояния через синтаксис автоматических свойств. Класс `Car` также имеет единственный абстрактный метод `TurboBoost()`, в котором применяется специальное перечисление (`EngineState`), представляющее текущее состояние двигателя автомобиля. Вставьте в проект новый файл класса C# по имени `Car.cs` со следующим кодом:

```
using System;

namespace CarLibrary
{
  // Абстрактный базовый класс в иерархии.
  public abstract class Car
  {
    public string PetName {get; set;}
    public int CurrentSpeed {get; set;}
    public int MaxSpeed {get; set;}

    protected EngineStateEnum State = EngineStateEnum.EngineAlive;
    public EngineStateEnum EngineState => State;
    public abstract void TurboBoost();

    protected Car(){}
    protected Car(string name, int maxSpeed, int currentSpeed)
    {
      PetName = name;
      MaxSpeed = maxSpeed;
      CurrentSpeed = currentSpeed;
    }
  }
}
```

Теперь предположим, что есть два непосредственных потомка класса `Car` с именами `MiniVan` (минивэн) и `SportsCar` (спортивный автомобиль). В каждом из них абстрактный метод `TurboBoost()` переопределяется для отображения подходящего сообщения в окне консоли. Вставьте в проект два новых файла классов C# с именами `MiniVan.cs` и `SportsCar.cs`. Поместите в них показанный ниже код:

```
// SportsCar.cs
using System;
namespace CarLibrary
{
  public class SportsCar : Car
  {
    public SportsCar(){ }
    public SportsCar(
      string name, int maxSpeed, int currentSpeed)
      : base (name, maxSpeed, currentSpeed){ }

    public override void TurboBoost()
    {
      Console.WriteLine("Ramming speed! Faster is better...");
    }
  }
}
```

```
// MiniVan.cs
using System;
namespace CarLibrary
{
  public class MiniVan : Car
  {
    public MiniVan(){ }
    public MiniVan(
      string name, int maxSpeed, int currentSpeed)
      : base (name, maxSpeed, currentSpeed){ }
    public override void TurboBoost()
    {
      // Минивэны имеют плохие возможности ускорения!
      State = EngineStateEnum.EngineDead;
      Console.WriteLine("Eek! Your engine block exploded!");
    }
  }
}
```

Исследование манифеста

Перед использованием `CarLibrary.dll` в клиентском приложении давайте посмотрим, как библиотека кода устроена внутри. Предполагая, что проект был скомпилирован, запустите утилиту `ildasm.exe` со скомпилированной сборкой. Если у вас нет утилиты `ildasm.exe` (описанной ранее в книге), то она также находится в каталоге для настоящей главы внутри хранилища GitHub.

```
ildasm /all /METADATA /out=CarLibrary.il .\CarLibrary\bin\Debug\
net5.0\CarLibrary.dll
```

Раздел манифеста `Manifest` дизассемблированных результатов начинается со строки `// Metadata version: 4.0.30319`. Непосредственно за ней следует список всех внешних сборок, требуемых для библиотеки классов:

```
// Metadata version: v4.0.30319
.assembly extern System.Runtime
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  .ver 5:0:0:0
}
.assembly extern System.Console
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  .ver 5:0:0:0
}
```

Каждый блок `.assembly extern` уточняется директивами `.publickeytoken` и `.ver`. Инструкция `.publickeytoken` присутствует только в случае, если сборка была сконфигурирована со *строгим именем*. Маркер `.ver` определяет числовой идентификатор версии ссылаемой сборки.

На заметку! Предшествующие версии .NET Framework в большой степени полагались на назначение строгих имен, которые вовлекали комбинацию открытого и секретного ключей. Это требовалось в среде Windows для сборок, подлежащих добавлению в глобальный кеш сборок, но с выходом .NET Core необходимость в строгих именах значительно снизилась.

После ссылок на внешние сборки вы обнаружите несколько маркеров `.custom`, которые идентифицируют атрибуты уровня сборки (кроме маркеров, сгенерированных системой, также информацию об авторском праве, название компании, версию сборки и т.д.). Ниже приведена (совсем) небольшая часть этой порции данных манифеста:

```
.assembly CarLibrary
{
  ...
  .custom instance void ... TargetFrameworkAttribute ...
  .custom instance void ... AssemblyCompanyAttribute ...
  .custom instance void ... AssemblyConfigurationAttribute ...
  .custom instance void ... AssemblyFileVersionAttribute ...
  .custom instance void ... AssemblyProductAttribute ...
  .custom instance void ... AssemblyTitleAttribute ...
```

Такие настройки могут устанавливаться либо с применением окна свойств проекта в Visual Studio, либо путем редактирования файла проекта и добавления надлежащих элементов. Находясь в среде Visual Studio, щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer, выберите в контекстном меню пункт Properties (Свойства) и перейдите не вкладку Package (Пакет) в левой части открывшегося диалогового окна (рис. 16.4).

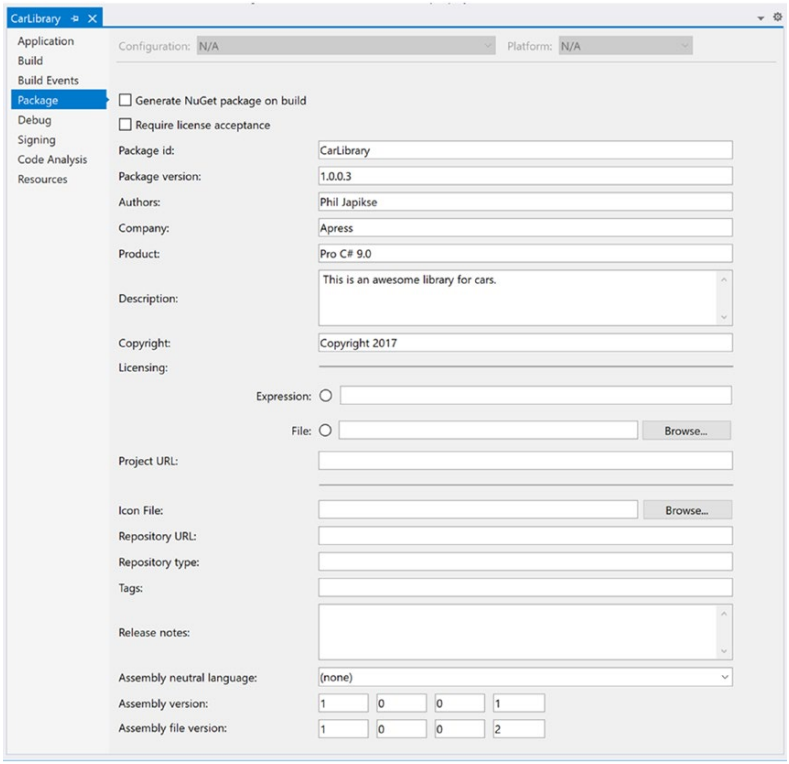

Рис. 16.4. Редактирование информации о сборке в окне свойств проекта в Visual Studio

Добавить метаданные к сборке можно и прямо в файле проекта *.csproj. Следующее обновление главного узла PropertyGroup в файле проекта приводит к тому же результату, что и заполнение формы, представленной на рис. 16.4:

```
<PropertyGroup>
  <TargetFramework>net5.0</TargetFramework>
  <Copyright>Copyright 2020</Copyright>
  <Authors>Phil Japikse</Authors>
  <Company>Apress</Company>
  <Product>Pro C# 9.0</Product>
  <PackageId>CarLibrary</PackageId>
  <Description>This is an awesome library for cars.</Description>
  <AssemblyVersion>1.0.0.1</AssemblyVersion>
  <FileVersion>1.0.0.2</FileVersion>
  <Version>1.0.0.3</Version>
</PropertyGroup>
```

На заметку! Остальные поля информации о сборке на рис. 16.4 (и в показанном выше содержимом файла проекта) используются при генерировании пакетов NuGet из вашей сборки. Данная тема раскрывается позже в главе.

Исследование кода CIL

Вспомните, что сборка не содержит инструкций, специфичных для платформы; взамен в ней хранятся инструкции на независимом от платформы общем промежуточном языке (Common Intermediate Language — CIL). Когда исполняющая среда .NET Core загружает сборку в память, ее внутренний код CIL компилируется (с использованием JIT-компилятора) в инструкции, воспринимаемые целевой платформой. Например, метод TurboBoost() класса SportsCar представлен следующим кодом CIL:

```
.method public hidebysig virtual
   instance void  TurboBoost() cil managed
{
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr "Ramming speed! Faster is better..."
  IL_0006:  call  void [System.Console]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  ret
}
// end of method SportsCar::TurboBoost
```

Большинству разработчиков приложений .NET Core нет необходимости глубоко погружаться в детали кода CIL. В главе 19 будут приведены дополнительные сведения о синтаксисе и семантике языка CIL, которые могут быть полезны при построении более сложных приложений, требующих расширенных действий вроде конструирования сборок во время выполнения.

Исследование метаданных типов

Прежде чем приступить к созданию приложений, в которых задействована ваша специальная библиотека .NET Core, давайте займемся исследованием метаданных для типов внутри сборки CarLibrary.dll. Скажем, вот определение TypeDef для типа EngineStateEnum:

```
TypeDef #1 (02000002)
-------------------------------------------------------
 TypDefName: CarLibrary.EngineStateEnum
 Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
 Extends   : [TypeRef] System.Enum
 Field #1
 -------------------------------------------------------
  Field Name: value__
  Flags     : [Public] [SpecialName] [RTSpecialName]
  CallCnvntn: [FIELD]
  Field type: I4
 Field #2
 -------------------------------------------------------
  Field Name: EngineAlive
  Flags     : [Public] [Static] [Literal] [HasDefault]
  DefltValue: (I4) 0
  CallCnvntn: [FIELD]
  Field type: ValueClass CarLibrary.EngineStateEnum
 Field #3
 -------------------------------------------------------
  Field Name: EngineDead
  Flags     : [Public] [Static] [Literal] [HasDefault]
  DefltValue: (I4) 1
  CallCnvntn: [FIELD]
  Field type: ValueClass CarLibrary.EngineStateEnum
```

Как будет объясняться в следующей главе, метаданные сборки являются важным элементом платформы .NET Core и служат основой для многочисленных технологий (сериализация объектов, позднее связывание, расширяемые приложения и т.д.). В любом случае теперь, когда вы заглянули внутрь сборки `CarLibrary.dll`, можно приступать к построению клиентских приложений, в которых будут применяться типы из сборки.

Построение клиентского приложения C#

Поскольку все типы в `CarLibrary` были объявлены с ключевым словом `public`, другие приложения .NET Core имеют возможность пользоваться ими. Вспомните, что типы могут также определяться с применением ключевого слова `internal` языка C# (на самом деле это стандартный режим доступа в C# для классов). Внутренние типы могут использоваться только в сборке, где они определены. Внешние клиенты не могут ни видеть, ни создавать экземпляры типов, помеченных ключевым словом `internal`.

На заметку! Исключением из указанного правила является ситуация, когда сборка явно разрешает доступ другой сборке с помощью атрибута `InternalsVisibleTo`, который вскоре будет рассмотрен.

Чтобы воспользоваться функциональностью вашей библиотеки, создайте в том же решении, где находится `CarLibrary`, новый проект консольного приложения C# по имени `CSharpCarClient`. Вы можете добиться цели с применением Visual Studio (щелкнув правой кнопкой мыши на имени решения и выбрав в контекстном меню

пункт Add⇨New Project (Добавить⇨Новый проект)) или командной строки (ниже показаны три команды, выполняемые по отдельности):

```
dotnet new console -lang c# -n CSharpCarClient -o .\CSharpCarClient -f net5.0
dotnet add CSharpCarClient reference CarLibrary
dotnet sln .\Chapter16_AppRojects.sln add .\CSharpCarClient
```

Приведенные команды создают проект консольного приложения, добавляют к нему ссылку на проект `CarLibrary` и вставляют его в имеющееся решение.

На заметку! Команда `add reference` создает ссылку на проект, что удобно на этапе разработки, т.к. `CSharpCarClient` будет всегда использовать последнюю версию `CarLibrary`. Можно также ссылаться прямо на сборку. Прямые ссылки создаются за счет указания скомпилированной библиотеки классов.

Если решение все еще открыто в Visual Studio, тогда вы заметите, новый проект отобразится в окне Solution Explorer безо всякого вмешательства с вашей стороны.

Наконец, щелкните правой кнопкой мыши на имени `CSharpCarClient` в окне Solution Explorer и выберите в контекстном меню пункт Set as Startup Project (Установить как стартовый проект). Если вы не работаете в Visual Studio, то можете запустить новый проект, введя команду `dotnet run` в каталоге проекта.

На заметку! Для установки ссылки на проект в Visual Studio можно также щелкнуть правой кнопкой мыши на имени проекта `CSharpCarClient` в окне Solution Explorer, выбрать в контекстном меню пункт Add⇨Reference (Добавить⇨Ссылка) и указать `CarLibrary` в узле проекта.

Теперь вы можете строить клиентское приложение для использования внешних типов. Модифицируйте начальный файл кода C#, как показано ниже:

```
using System;
// Не забудьте импортировать пространство имен CarLibrary!
using CarLibrary;

Console.WriteLine("***** C# CarLibrary Client App *****");

// Создать объект SportsCar.
SportsCar viper = new SportsCar("Viper", 240, 40);
viper.TurboBoost();

// Создать объект MiniVan.
MiniVan mv = new MiniVan();
mv.TurboBoost();

Console.WriteLine("Done. Press any key to terminate");
                  // Готово. Нажмите любую клавишу для прекращения работы
Console.ReadLine();
```

Код выглядит очень похожим на код в других приложениях, которые разрабатывались в книге ранее. Единственный интересный аспект связан с тем, что в клиентском приложении C# теперь применяются типы, определенные внутри отдельной специальной библиотеки. Запустив приложение, можно наблюдать отображение разнообразных сообщений.

Вас может интересовать, что в точности происходит при ссылке на проект `CarLibrary`. Когда создается *ссылка на проект*, порядок компиляции решения кор-

ректируется таким образом, чтобы зависимые проекты (CarLibrary в рассматриваемом примере) компилировались первыми и результат компиляции копировался в выходной каталог родительского проекта (CSharpCarLibrary). Скомпилированная клиентская библиотека ссылается на скомпилированную библиотеку классов. При повторной компиляции клиентского проекта то же самое происходит и с зависимой библиотекой, так что новая версия снова копируется в целевой каталог.

На заметку! Если вы используете Visual Studio, то можете щелкнуть на кнопке Show All Files (Показать все файлы) в окне Solution Explorer, что позволит увидеть все выходные файлы и удостовериться в наличии там скомпилированной библиотеки CarLibrary. Если вы работаете в Visual Studio Code, тогда перейдите в каталог bin\debug\net5.0 на вкладке Explorer (Проводник).

Когда создается *прямая ссылка*, скомпилированная библиотека тоже копируется в выходной каталог клиентской библиотеки, но во время создания ссылки. Без ссылки на проект сами проекты можно компилировать независимо друг от друга и файлы могут стать несогласованными. Выражаясь кратко, если вы разрабатываете зависимые библиотеки (как обычно происходит в реальных программных проектах), то лучше ссылаться на проект, а не на результат компиляции проекта.

Построение клиентского приложения Visual Basic

Вспомните, что платформа .NET Core позволяет разработчикам разделять скомпилированный код между языками программирования. Чтобы проиллюстрировать языковую независимость платформы .NET Core, создайте еще один проект консольного приложения (по имени VisualBasicCarClient) на этот раз с применением языка Visual Basic (имейте в виду, что каждая команда вводится в отдельной строке):

```
dotnet new console -lang vb -n VisualBasicCarClient
                   -o .\VisualBasicCarClient -f net5.0
dotnet add VisualBasicCarClient reference CarLibrary
dotnet sln .\Chapter16_AllProjects.sln add VisualBasicCarClient
```

Подобно C# язык Visual Basic позволяет перечислять все пространства имен, используемые внутри текущего файла. Тем не менее, вместо ключевого слова using, применяемого в C#, для такой цели в Visual Basic служит ключевое слово Imports, поэтому добавьте в файл кода Program.vb следующий оператор Imports:

```
Imports CarLibrary
Module Program
  Sub Main()
  End Sub
End Module
```

Обратите внимание, что метод Main() определен внутри типа модуля Visual Basic. По существу модули представляют собой систему обозначений Visual Basic для определения класса, который может содержать только статические методы (очень похоже на статический класс C#). Итак, чтобы испробовать типы MiniVan и SportsCar, используя синтаксис Visual Basic, модифицируйте метод Main(), как показано ниже:

```
Sub Main()
  Console.WriteLine("***** VB CarLibrary Client App *****")
  ' Локальные переменные объявляются с применением ключевого слова Dim.
```

```
  Dim myMiniVan As New MiniVan()
  myMiniVan.TurboBoost()

  Dim mySportsCar As New SportsCar()
  mySportsCar.TurboBoost()
  Console.ReadLine()
End Sub
```

После компиляции и запуска приложения (не забудьте установить `VisualBasic CarClient` как стартовый проект в Visual Studio) снова отобразится последовательность окон с сообщениями. Кроме того, новое клиентское приложение имеет собственную локальную копию `CarLibrary.dll` в своем каталоге `bin\Debug\net5.0`.

Межъязыковое наследование в действии

Привлекательным аспектом разработки в .NET Core является понятие *межъязыкового наследования*. В целях иллюстрации давайте создадим новый класс Visual Basic, производный от типа `SportsCar` (который был написан на C#). Для начала добавьте в текущее приложение Visual Basic новый файл класса по имени `PerformanceCar.vb`. Модифицируйте начальное определение класса, унаследовав его от типа `SportsCar` с применением ключевого слова `Inherits`. Затем переопределите абстрактный метод `TurboBoost()`, используя ключевое слово `Overrides`:

```
Imports CarLibrary

' Этот класс VB унаследован от класса SportsCar, написанного на C#.
Public Class PerformanceCar
  Inherits SportsCar

  Public Overrides Sub TurboBoost()
    Console.WriteLine("Zero to 60 in a cool 4.8 seconds...")
  End Sub
End Class
```

Чтобы протестировать новый тип класса, модифицируйте код метода `Main()` в модуле:

```
Sub Main()
  ...
  Dim dreamCar As New PerformanceCar()

  ' Использовать унаследованное свойство.
  dreamCar.PetName = "Hank"
  dreamCar.TurboBoost()
  Console.ReadLine()
End Sub
```

Обратите внимание, что объект `dreamCar` способен обращаться к любому открытому члену (такому как свойство `PetName`), расположенному выше в цепочке наследования, невзирая на тот факт, что базовый класс был определен на совершенно другом языке и находится полностью в другой сборке! Возможность расширения классов за пределы границ сборок в независимой от языка манере — естественный аспект цикла разработки в .NET Core. Он упрощает применение скомпилированного кода, написанного программистами, которые предпочли не создавать свой разделяемый код на языке C#.

Открытие доступа к внутренним типам для других сборок

Как упоминалось ранее, внутренние (`internal`) классы видимы остальным объектам только в сборке, где они определены. Исключением является ситуация, когда видимость явно предоставляется другому проекту.

Начните с добавления в проект `CarLibrary` нового класса по имени `MyInternal Class` со следующим кодом:

```
namespace CarLibrary
{
  internal class MyInternalClass
  {
  }
}
```

На заметку! Зачем вообще открывать доступ к внутренним типам? Обычно это делается для модульного и интеграционного тестирования. Разработчики хотят иметь возможность тестировать свой код, но не обязательно открывать к нему доступ за границами сборки.

Использование атрибута `assembly`

Атрибуты будут более детально раскрыты в главе 17, но пока откройте файл класса `Car.cs` из проекта `CarLibrary` и добавьте показанный ниже атрибут и оператор `using`:

```
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("CSharpCarClient")]
namespace CarLibrary
{
}
```

Атрибут `InternalsVisibleTo` принимает имя проекта, который может видеть класс с установленным атрибутом. Имейте в виду, что другие проекты не в состоянии "запрашивать" такое разрешение; оно должно быть предоставлено проектом, содержащим внутренние типы.

На заметку! В предшествующих версиях .NET использовался файл класса `AssemblyInfo.cs`, который по-прежнему существует в .NET Core, но генерируется автоматически и не предназначен для потребления разработчиками.

Теперь можете модифицировать проект `CSharpCarClient`, добавив в метод `Main()` следующий код:

```
var internalClassInstance = new MyInternalClass();
```

Код работает нормально. Затем попробуйте сделать то же самое в методе `Main()` проекта `VisualBasicCarClient`:

```
' Не скомпилируется.
' Dim internalClassInstance = New MyInternalClass()
```

Поскольку библиотека `VisualBasicCarClient` не предоставила разрешение видеть внутренние типы, предыдущий код не скомпилируется.

Использование файла проекта

Еще один способ добиться того же (и можно утверждать, что он в большей степени соответствует стилю .NET Core) предусматривает применение обновленных возможностей файла проекта .NET Core.

Закомментируйте только что добавленный атрибут и откройте файл проекта `CarLibrary`. Добавьте в файл проекта узел `ItemGroup`, как показано ниже:

```
<ItemGroup>
  <AssemblyAttribute
Include="System.Runtime.CompilerServices.InternalsVisibleToAttribute">
    <_Parameter1>CSharpCarClient</_Parameter1>
  </AssemblyAttribute>
</ItemGroup>
```

Результат оказывается таким же, как в случае использования атрибута в классе, и считается более удачным решением, потому что другие разработчики будут видеть его прямо в файле проекта, а не искать повсюду в проекте.

NuGet и .NET Core

NuGet — это диспетчер пакетов для .NET и .NET Core. Он является механизмом для совместного использования программного обеспечения в формате, который воспринимается приложениями .NET Core, а также стандартным способом загрузки .NET Core и связанных инфраструктур (ASP.NET Core, EF Core и т.д.). Многие организации помещают в пакеты NuGet свои стандартные сборки, предназначенные для решения сквозных задач (наподобие ведения журнала и построения отчетов об ошибках), с целью потребления в разрабатываемых бизнес-приложениях.

Пакетирование сборок с помощью NuGet

Чтобы увидеть пакетирование в действии, понадобиться поместить библиотеку `CarLibrary` внутрь пакета и затем ссылаться на пакет из двух клиентских приложений.

Свойства пакета NuGet доступны через окно свойств проекта. Щелкните правой кнопкой мыши на имени проекта `CarLibrary` и выберите в контекстном меню пункт Properties (Свойства). Перейдя на вкладку Package (Пакет), вы увидите значения, которые вводились ранее для настройки сборки. Для пакета NuGet можно установить дополнительные свойства (скажем, принятие лицензионного соглашения и информацию о проекте, такую как URL и местоположение хранилища).

На заметку! Все значения на вкладке Package пользовательского интерфейса Visual Studio могут быть введены в файле проекта вручную, но вы должны знать ключевые слова. Имеет смысл хотя бы раз воспользоваться Visual Studio для ввода всех значений и затем вручную редактировать файл проекта. Кроме того, все допустимые свойства описаны в документации по .NET Core.

В текущем примере кроме флажка Generate NuGet package on build (Генерировать пакет NuGet при компиляции) никаких дополнительных свойств устанавливать не нужно. Можно также модифицировать файл проекта следующим образом:

```
<PropertyGroup>
  <TargetFramework>net5.0</TargetFramework>
  <Copyright>Copyright 2020</Copyright>
  <Authors>Phil Japikse</Authors>
  <Company>Apress</Company>
  <Product>Pro C# 9.0</Product>
  <PackageId>CarLibrary</PackageId>
  <Description>This is an awesome library for cars.</Description>
  <AssemblyVersion>1.0.0.1</AssemblyVersion>
  <FileVersion>1.0.0.2</FileVersion>
  <Version>1.0.0.3</Version>
  <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
</PropertyGroup>
```

Это приведет к тому, что пакет будет создаваться заново при каждой компиляции проекта. По умолчанию пакет создается в подкаталоге `bin\Debug` или `bin\Release` в зависимости от выбранной конфигурации.

Пакеты также можно создавать в командной строке, причем интерфейс CLI предлагает больше параметров, чем среда Visual Studio. Например, чтобы построить пакет и поместить его в каталог по имени `Publish`, введите показанные далее команды (находясь в каталоге проекта `CarLibrary`). Первая команда компилирует сборку, а вторая создает пакет NuGet.

```
dotnet build -c Release
dotnet pack -o .\Publish -c Debug
```

На заметку! `Debug` является стандартной конфигурацией и потому указывать `-c Debug` необязательно, но параметр присутствует в команде, чтобы намерение стало совершенно ясным.

Теперь в каталоге `Publish` находится файл `CarLibrary.1.0.0.3.nupkg`. Для просмотра его содержимого откройте файл с помощью любой утилиты zip-архивации (такой как 7-Zip). Вы увидите полное содержимое, которое включает сборку и дополнительные метаданные.

Ссылка на пакеты NuGet

Вас может интересовать, откуда поступают пакеты, добавленные в предшествующих примерах. Местоположением пакетов NuGet управляет файл XML по имени `NuGet.Config`. В среде Windows он находится в каталоге `%appdata%\NuGet`. Это главный файл. Открыв его, вы увидите несколько источников пакетов:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
         protocolVersion="3" />
    <add key="Microsoft Visual Studio Offline Packages"
       value="C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\" />
  </packageSources>
</configuration>
```

Здесь присутствуют два источника пакетов. Первый источник указывает на `http://nuget.org/` — крупнейшее в мире хранилище пакетов NuGet. Второй источник находится на вашем локальном диске и применяется средой Visual Studio в качестве кеша пакетов.

Важно отметить, что файлы `NuGet.Config` по умолчанию являются *аддитивными*. Чтобы добавить дополнительные источники, не изменяя список источников для всей системы, вы можете создавать дополнительные файлы `NuGet.Config`. Каждый файл действителен для каталога, в котором он находится, а также для любых имеющихся подкаталогов. Добавьте в каталог решения новый файл по имени `NuGet.Config` со следующим содержимым:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="local-packages" value=".\CarLibrary\Publish" />
  </packageSources>
</configuration>
```

Кроме того, вы можете очищать список источников пакетов, добавляя в узел `<packageSources>` элемент `<clear />`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <clear />
    <add key="local-packages" value=".\CarLibrary\Publish" />
    <add key="NuGet" value="https://api.nuget.org/v3/index.json" />
  </packageSources>
</configuration>
```

На заметку! В случае работы в Visual Studio вам придется перезапустить IDE-среду, чтобы обновленные настройки `NuGet.Config` вступили в силу.

Удалите ссылки на проекты из проектов `CSharpCarClient` и `VisualBasic CarClient`, после чего добавьте ссылки на пакет (находясь в каталоге решения):

```
dotnet add CSharpCarClient package CarLibrary
dotnet add VisualBasicCarClient package CarLibrary
```

Установив ссылки, скомпилируйте решение и просмотрите целевой каталог (`bin\Debug\new5.0`). Вы увидите, что в целевом каталоге находится файл `CarLibrary.dll`, а файл `CarLibrary.nupkg` отсутствует. Причина в том, что исполняющая среда .NET Core распаковывает файл `CarLibrary.nupkg` и добавляет содержащиеся в нем сборки как прямые ссылки.

Установите одного из клиентских проектов в качестве стартового и запустите приложение; оно будет функционировать точно так же, как ранее.

Смените номер версии библиотеки `CarLibrary` на 1.0.0.4 и снова создайте пакет. Теперь в каталоге `Publish` присутствуют два NuGet-пакета `CarLibrary`. Если вы опять выполните команды `add package`, то проект обновится для использования новой версии. На тот случай, когда предпочтительнее более старая версия, команда `add package` позволяет добавить номер версии для определенного пакета.

Опубликование консольных приложений (обновление в версии .NET 5)

Итак, имея приложение `CarClient` на C# (и связанную с ним сборку `CarLibrary`), каким образом вы собираетесь передавать его своим пользователям? Пакетирование приложения вместе с его зависимостями называется *опубликованием*. Опубликование приложений .NET Framework требовало, чтобы на целевой машине была установлена инфраструктура, и приложения .NET Core также могут быть опубликованы похожим способом, который называется развертыванием, *зависящим от инфраструктуры*. Однако приложения .NET Core вдобавок могут публиковаться как *автономные*, которые вообще не требуют наличия установленной платформы .NET Core! Когда приложение публикуется как автономное, вы обязаны указать идентификатор целевой исполняющей среды. Идентификатор исполняющей среды применяется для пакетирования вашего приложения, ориентированного на определенную ОС. Полный список доступных идентификаторов исполняющих сред приведен в каталоге .NET Core RID Catalog по ссылке `https://docs.microsoft.com/ru-ru/dotnet/core/rid-catalog`.

На заметку! Опубликование приложений ASP.NET Core — более сложный процесс, который будет раскрыт позже в книге.

Опубликование приложений, зависящих от инфраструктуры

Развертывание, зависящее от инфраструктуры, представляет собой стандартный режим для команды `dotnet publish`. Чтобы создать пакет с вашим приложением и обязательными файлами, понадобится лишь выполнить следующую команду в интерфейсе командной строки:

```
dotnet publish
```

На заметку! Команда `publish` использует стандартную конфигурацию для вашего проекта, которой обычно является `Debug`.

Приведенная выше команда помещает ваше приложение и поддерживающие его файлы (всего 16 файлов) в каталог `bin\Debug\net5.0\publish`. Заглянув в упомянутый каталог, вы обнаружите два файла `*.dll` (`CarLibrary.dll` и `CSharpCarClient.dll`), которые содержат весь прикладной код. В качестве напоминания: файл `CSharpCarClient.exe` представляет собой пакетированную версию `dotnet.exe`, сконфигурированную для запуска `CSharpCarClient.dll`. Дополнительные файлы в каталоге — это файлы .NET Core, которые не входят в состав .NET Core Runtime.

Чтобы создать версию `Release` (которая будет помещена в каталог `bin\release\net5.0\publish`), введите такую команду:

```
dotnet publish -c release
```

Опубликование автономных приложений

Подобно развертыванию, зависящему от инфраструктуры, автономное развертывание включает весь прикладной код и сборки, на которые производилась ссылка, а также файлы .NET Core Runtime, требующиеся приложению. Чтобы опубликовать свое приложение как автономное развертывание, выполните следующую команду CLI (указывающую в качестве выходного местоположения каталог по имени `selfcontained`):

```
dotnet publish -r win-x64 -c release -o selfcontained --self-contained true
```

На заметку! При создании автономного развертывания обязателен идентификатор исполняющей среды, чтобы процессу опубликования было известно, какие файлы .NET Core Runtime добавлять к вашему прикладному коду.

Команда помещает ваше приложение и его поддерживающие файлы (всего 235 файлов) в каталог `selfcontained`. Если вы скопируете эти файлы на другой компьютер с 64-разрядной ОС Windows, то сможете запускать приложение, даже если исполняющая среда .NET 5 на нем не установлена.

Опубликование автономных приложений в виде единственного файла

В большинстве ситуаций развертывание 235 файлов (для приложения, которое выводит всего лишь несколько строк текста) вряд ли следует считать наиболее эффективным способом предоставления вашего приложения пользователям. К счастью, в .NET 5 значительно улучшена возможность опубликования вашего приложения и межплатформенных файлов исполняющей среды в виде единственного файла. Не включаются только файлы собственных библиотек, которые должны существовать вне одиночного файла EXE.

Показанная ниже команда создает однофайловое автономное развертывание для 64-разрядных ОС Windows и помещает результат в каталог по имени `singlefile`:

```
dotnet publish -r win-x64 -c release -o singlefile
      --self-contained true -p:PublishSingleFile=true
```

Исследуя файлы, которые были созданы, вы обнаружите один исполняемый файл (`CSharpCarClient.exe`), отладочный файл (`CSharpCarClient.pdb`) и четыре DLL-библиотеки, специфичные для ОС. В то время как предыдущий процесс опубликования производил 235 файлов, однофайловая версия `CSharpCarClient.exe` имеет размер 54 Мбайт! Создание однофайлового развертывания упаковывает 235 файлов в единственный файл. За снижение количества файлов приходится платить увеличением размера файла.

Напоследок важно отметить, что собственные библиотеки тоже можно поместить в единственный файл. Модифицируйте файл `CSharpCarClient.csproj` следующим образом:

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="CarLibrary" Version="1.0.0.3" />
  </ItemGroup>
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <IncludeNativeLibrariesForSelfExtract>true
    </IncludeNativeLibrariesForSelfExtract>
  </PropertyGroup>
</Project>
```

После запуска приведенной выше команды `dotnet publish` на выходе окажется одиночный файл. Тем не менее, это только механизм транспортировки. При запуске приложения файлы собственных библиотек будут извлечены во временное местоположение на целевой машине.

Определение местонахождения сборок исполняющей средой .NET Core

Все сборки, построенные до сих пор, были связаны напрямую (кроме только что законченного примера с пакетом NuGet). Вы добавляли либо ссылку на проект, либо прямую ссылку между проектами. В таких случаях (и в примере с NuGet) зависимая сборка копировалась напрямую в целевой каталог клиентского приложения. Определение местонахождения зависимой сборки не является проблемой, т.к. она размещается на диске рядом с приложением, которое в ней нуждается.

Но что насчет инфраструктуры .NET Core? Как ищутся ее сборки? В предшествующих версиях .NET файлы инфраструктуры устанавливались в глобальный кеш сборок (GAC), так что всем приложениям .NET было известно, где их найти.

Однако GAC мешает реализовать возможности параллельного выполнения разных версий приложений в .NET Core, поэтому здесь нет одиночного хранилища для файлов исполняющей среды и инфраструктуры. Взамен все файлы, составляющие инфраструктуру, устанавливаются в каталог `C:\Program Files\dotnet` (в среде Windows) с разделением по версиям. В зависимости от версии приложения (как указано в файле `.csproj`) необходимые файлы исполняющей среды и инфраструктуры загружаются из каталога заданной версии.

В частности, при запуске какой-то версии исполняющей среды предоставляется набор *путей зондирования*, которые будут применяться для нахождения зависимостей приложения. Существуют пять свойств зондирования, перечисленные в табл. 16.1 (все они необязательны).

Таблица 16.1. Свойства зондирования приложений

Свойство	Описание
`TRUSTED_PLATFORM_ASSEMBLIES`	Список путей к файлам сборок платформы и приложения
`PLATFORM_RESOURCE_ROOTS`	Список путей к каталогам для поиска подчиненных сборок ресурсов
`NATIVE_DLL_SEARCH_DIRECTORIES`	Список путей к каталогам для поиска неуправляемых (собственных) библиотек
`APP_PATHS`	Список путей к каталогам для поиска управляемых сборок
`APP_NI_PATHS`	Список путей к каталогам для поиска собственных образов управляемых сборок

Чтобы выяснить стандартные пути зондирования, создайте новый проект консольного приложения .NET Core по имени `FunWithProbingPaths`. Приведите операторы верхнего уровня к следующему виду:

```
using System;
using System.Linq;
Console.WriteLine("*** Fun with Probing Paths ***");
Console.WriteLine($"TRUSTED_PLATFORM_ASSEMBLIES: ");
// Для платформ, отличающихся от Windows, используйте ':'.
var list = AppContext.GetData("TRUSTED_PLATFORM_ASSEMBLIES")
                              .ToString().Split(';');
foreach (var dir in list)
{
```

```
    Console.WriteLine(dir);
  }
  Console.WriteLine();
  Console.WriteLine($"PLATFORM_RESOURCE_ROOTS:
{AppContext.GetData ("PLATFORM_RESOURCE_ROOTS")}");
  Console.WriteLine();
  Console.WriteLine($"NATIVE_DLL_SEARCH_DIRECTORIES:
{AppContext.GetData ("NATIVE_DLL_SEARCH_DIRECTORIES")}");
  Console.WriteLine();
  Console.WriteLine($"APP_PATHS: {AppContext.GetData("APP_PATHS")}");
  Console.WriteLine();
  Console.WriteLine($"APP_NI_PATHS: {AppContext.GetData("APP_NI_PATHS")}");
  Console.WriteLine();
  Console.ReadLine();
```

Запустив приложекние, вы увидите большинство значений, поступающих из переменной `TRUSTED_PLATFORM_ASSEMBLIES`. В дополнение к сборке, созданной для этого проекта в целевом каталоге, будет выведен список библиотек базовых классов из каталога текущей исполняющей среды, `C:\Program Files\dotnet\shared\Microsoft.NETCore.App\5.0.0` (номер версии у вас может быть другим).

В список добавляется каждый файл, на который напрямую ссылается ваше приложение, а также любые файлы исполняющей среды, требующиеся вашему приложению. Список библиотек исполняющей среды заполняется одним или большим числом файлов `*.deps.json`, которые загружаются вместе с исполняющей средой .NET Core. Они находятся в каталоге установки для комплекта SDK (применяется при построении программ) и исполняющей среды (используется при выполнении программ). В рассматриваемом простом примере задействован только один файл такого рода — `Microsoft.NETCore.App.deps.json`.

По мере возрастания сложности вашего приложения будет расти и список файлов в `TRUSTED_PLATFORM_ASSEMBLIES`. Скажем, если вы добавите ссылку на пакет `Microsoft.EntityFrameworkCore`, то список требующихся сборок расширится. Чтобы удостовериться в этом, введите показанную ниже команду в консоли диспетчера пакетов (в каталоге, где располагается файл `*.csproj`):

```
dotnet add package Microsoft.EntityFrameworkCore
```

После добавления пакета снова запустите приложение и обратите внимание, насколько больше стало файлов в списке. Хотя вы добавили только одну новую ссылку, пакет `Microsoft.EntityFrameworkCore` имеет собственные зависимости, которые добавляются в `TRUSTED_PLATFORM_ASSEMBLIES`.

Резюме

В главе была исследована роль библиотек классов .NET Core (файлов `*.dll`). Вы видели, что библиотеки классов представляют собой двоичные файлы .NET Core, содержащие логику, которая предназначена для многократного использования в разнообразных проектах.

Вы ознакомились с деталями разнесения типов по пространствам имен .NET Core и отличием между .NET Core и .NET Standard, приступили к конфигурированию приложений и углубились в состав библиотек классов. Затем вы научились публиковать консольные приложения .NET Core. В заключение вы узнали, каким образом пакетировать свои приложения с применением NuGet.

ГЛАВА 17

Рефлексия типов, позднее связывание и программирование на основе атрибутов

Как было показано в главе 16, сборки являются базовой единицей развертывания в мире .NET Core. Используя интегрированный браузер объектов Visual Studio (и многих других IDE-сред), можно просматривать типы внутри набора сборок, на которые ссылается проект. Кроме того, внешние инструменты, такие как утилита `ildasm.exe`, позволяют заглядывать внутрь лежащего в основе кода CIL, метаданных типов и манифеста сборки для заданного двоичного файла .NET Core. В дополнение к подобному исследованию сборок .NET Core на этапе проектирования ту же самую информацию можно получить *программно* с применением пространства имен `System.Reflection`. Таким образом, первой задачей настоящей главы является определение роли рефлексии и потребности в метаданных .NET Core.

Остаток главы посвящен нескольким тесно связанным темам, которые вращаются вокруг служб рефлексии. Например, вы узнаете, как клиент .NET Core может задействовать динамическую загрузку и позднее связывание для активизации типов, сведения о которых на этапе компиляции отсутствуют. Вы также научитесь вставлять специальные метаданные в сборки .NET Core за счет использования системных и специальных атрибутов. Для практической демонстрации всех этих аспектов в завершение главы приводится пример построения нескольких "объектов-оснасток", которые можно подключать к расширяемому консольному приложению.

Потребность в метаданных типов

Возможность полного описания типов (классов, интерфейсов, структур, перечислений и делегатов) с помощью метаданных является ключевым элементом платформы .NET Core. Многим технологиям .NET Core, таким как сериализация объектов, требуется способность выяснения формата типов во время выполнения. Кроме того, межъязыковое взаимодействие, многие службы компилятора и средства IntelliSense в IDE-среде опираются на конкретное описание *типа*.

Вспомните, что утилита `ildasm.exe` позволяет просматривать метаданные типов в сборке. Чтобы взглянуть на метаданные сборки `CarLibrary`, перейдите к разделу `METAINFO` в сгенерированном файле `CarLibrary.il` (из главы 16). Ниже приведен небольшой их фрагмент:

```
// ==== M E T A I N F O ===
// ============================================================
// ScopeName : CarLibrary.dll
// MVID      : {598BC2B8-19E9-46EF-B8DA-672A9E99B603}
// ============================================================
// Global functions
// ------------------------------------------------------------
//
// Global fields
// ------------------------------------------------------------
//
// Global MemberRefs
// ------------------------------------------------------------
//
// TypeDef #1
// ------------------------------------------------------------
//   TypDefName: CarLibrary.Car
//   Flags     : [Public] [AutoLayout] [Class] [Abstract] [AnsiClass]
[BeforeFieldInit]
//   Extends   : [TypeRef] System.Object
//   Field #1
//   ------------------------------------------------------------
//     Field Name: value__
//     Flags     : [Private]
//     CallCnvntn: [FIELD]
//     Field type: String
//
```

Как видите, утилита `ildasm.exe` отображает метаданные типов .NET Core очень подробно (фактический двоичный формат гораздо компактнее). В действительности описание всех метаданных сборки `CarLibrary.dll` заняло бы несколько страниц. Однако для понимания вполне достаточно кратко взглянуть на некоторые ключевые описания метаданных сборки `CarLibrary.dll`.

На заметку! Не стоит слишком глубоко вникать в синтаксис каждого фрагмента метаданных .NET Core, приводимого в нескольких последующих разделах. Важно усвоить, что метаданные .NET Core являются исключительно описательными и учитывают каждый внутренне определенный (и внешне ссылаемый) тип, который найден в имеющейся кодовой базе.

Просмотр (частичных) метаданных для перечисления `EngineStateEnum`

Каждый тип, определенный внутри текущей сборки, документируется с применением маркера `TypeDef #n` (где `TypeDef` — сокращение от *type definition* (определение типа)). Если описываемый тип использует какой-то тип, определенный в отдельной сборке .NET Core, тогда ссылаемый тип документируется с помощью маркера `TypeRef #n` (где `TypeRef` — сокращение от *type reference* (ссылка на тип)). Если хотите, то можете считать, что маркер `TypeRef` является указателем на полное опреде-

ление метаданных ссылаемого типа во внешней сборке. Коротко говоря, метаданные .NET Core — это набор таблиц, явно помечающих все определения типов (TypeDef) и ссылаемые типы (TypeRef), которые могут быть просмотрены с помощью утилиты ildasm.exe.

В случае сборки CarLibrary.dll один из маркеров TypeDef представляет описание метаданных перечисления CarLibrary.EngineStateEnum (номер TypeDef у вас может отличаться; нумерация TypeDef основана на порядке, в котором компилятор C# обрабатывает файл):

```
// TypeDef #2
// ------------------------------------------------------
//   TypDefName: CarLibrary.EngineStateEnum
//   Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
//   Extends   : [TypeRef] System.Enum
//   Field #1
//   -------------------------------------------------------
//     Field Name: value__
//     Flags     : [Public] [SpecialName] [RTSpecialName]
//     CallCnvntn: [FIELD]
//     Field type: I4
//
//   Field #2
//   -------------------------------------------------------
//     Field Name: EngineAlive
//     Flags     : [Public] [Static] [Literal] [HasDefault]
//     DefltValue: (I4) 0
//     CallCnvntn: [FIELD]
//     Field type: ValueClass CarLibrary.EngineStateEnum
//
...
```

Маркер TypDefName служит для установления имени заданного типа, которым в рассматриваемом случае является специальное перечисление CarLibrary.EngineStateEnum. Маркер метаданных Extends применяется при документировании базового типа для заданного типа .NET Core (ссылаемого типа System.Enum в этом случае). Каждое поле перечисления помечается с использованием маркера Field #n. Ради краткости выше была приведена только часть метаданных.

На заметку! Хотя это выглядит как опечатка, в TypDefName отсутствует буква "e", которую можно было бы ожидать.

Просмотр (частичных) метаданных для типа Car

Ниже показана часть метаданных класса Car, которая иллюстрирует следующие аспекты:

- как поля определяются в терминах метаданных .NET Core;
- как методы документируются посредством метаданных .NET Core;
- как автоматическое свойство представляется в метаданных .NET Core.

```
   // TypeDef #1
   // --------------------------------------------------------
   //   TypDefName: CarLibrary.Car
   //   Flags     : [Public] [AutoLayout] [Class] [Abstract] [AnsiClass]
[BeforeFieldInit]
   //   Extends   : [TypeRef] System.Object
   //   Field #1
   //   -------------------------------------------------------
   //     Field Name: <PetName>k__BackingField
   //     Flags     : [Private]
   //     CallCnvntn: [FIELD]
   //     Field type: String
   ...
   //   Method #1
   //   -------------------------------------------------------
   //     MethodName: get_PetName
   //     Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]
   //     RVA       : 0x000020d0
   //     ImplFlags : [IL] [Managed]
   //     CallCnvntn: [DEFAULT]
   //     hasThis
   //     ReturnType: String
   //     No arguments.
   ...
   //   Method #2
   //   -------------------------------------------------------
   //     MethodName: set_PetName
   //     Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]
   //     RVA       : 0x00002058
   //     ImplFlags : [IL] [Managed]
   //     CallCnvntn: [DEFAULT]
   //     hasThis
   //     ReturnType: Void
   //     1 Arguments
   //       Argument #1: String
   //     1 Parameters
   //       (1) ParamToken : Name : value flags: [none]
   ...
   //   Property #1
   //   -------------------------------------------------------
   //     Prop.Name : PetName
   //     Flags     : [none]
   //     CallCnvntn: [PROPERTY]
   //     hasThis
   //     ReturnType: String
   //     No arguments.
   //     DefltValue:
   //     Setter    : set_PetName
   //     Getter    : get_PetName
   //     0 Others
   ...
```

Прежде всего, метаданные класса `Car` указывают базовый класс этого типа (`System.Object`) и включают разнообразные флаги, которые описывают то, как тип был сконструирован (например, `[Public]`, `[Abstract]` и т.п.). Описания методов (вроде конструктора `Car`) содержат имя, возвращаемое значение и параметры.

Обратите внимание, что автоматическое свойство дает в результате сгенерированное компилятором закрытое поддерживающее поле (по имени `<PetName>k__BackingField`) и два сгенерированных компилятором метода (в случае свойства для чтения и записи) с именами `get_PetName()` и `set_PetName()`. Наконец, само свойство отображается на внутренние методы получения/установки с применением маркеров `Setter` и `Getter` метаданных .NET Core.

Исследование блока TypeRef

Вспомните, что метаданные сборки будут описывать не только набор внутренних типов (`Car`, `EngineStateEnum` и т.д.), но также любые внешние типы, на которые ссылаются внутренние типы. Например, с учетом того, что в сборке `CarLibrary.dll` определены два перечисления, метаданные типа `System.Enum` будут содержать следующий блок `TypeRef`:

```
// TypeRef #19
// -------------------------------------------------------
//   Token           : 0x01000013
//   ResolutionScope: 0x23000001
//   TypeRefName     : System.Enum
```

Документирование определяемой сборки

В файле `CarLibrary.il` также присутствуют метаданные .NET Core, которые описывают саму сборку с использованием маркера `Assembly`. Ниже представлена часть метаданных манифеста сборки `CarLibrary.dll`:

```
// Assembly
// -------------------------------------------------------
//   Token: 0x20000001
//   Name : CarLibrary
//   Public Key :
//   Hash Algorithm : 0x00008004
//   Version: 1.0.0.1
//   Major Version: 0x00000001
//   Minor Version: 0x00000000
//   Build Number: 0x00000000
//   Revision Number: 0x00000001
//   Locale: <null>
//   Flags : [none] (00000000)
```

Документирование ссылаемых сборок

В дополнение к маркеру `Assembly` и набору блоков `TypeDef` и `TypeRef` в метаданных .NET Core также применяются маркеры `AssemblyRef #n` для документирования каждой внешней сборки. С учетом того, что каждая сборка .NET Core ссылается на библиотеку базовых классов `System.Runtime`, вы обнаружите `AssemblyRef` для сборки `System.Runtime`, как показано в следующем фрагменте:

```
// AssemblyRef #1 (23000001)
// -------------------------------------------------------
//   Token: 0x23000001
//   Public Key or Token: b0 3f 5f 7f 11 d5 0a 3a
//   Name: System.Runtime
//   Version: 5.0.0.0
//   Major Version: 0x00000005
//   Minor Version: 0x00000000
//   Build Number: 0x00000000
//   Revision Number: 0x00000000
//   Locale: <null>
//   HashValue Blob:
//   Flags: [none] (00000000)
```

Документирование строковых литералов

Последний полезный аспект, относящийся к метаданным .NET Core, связан с тем, что все строковые литералы в кодовой базе документируются внутри маркера `User Strings`:

```
// User Strings
// -------------------------------------------------------
//   70000001 : (23) L"CarLibrary Version 2.0!"
//   70000031 : (13) L"Quiet time..."
//   7000004d : (11) L"Jamming {0}"
//   70000065 : (32) L"Eek! Your engine block exploded!"
//   700000a7 : (34) L"Ramming speed! Faster is better..."
```

На заметку! Всегда помните о том, что все строки явным образом документируются в метаданных сборки, как продемонстрировано в представленном выше листинге метаданных. Это может привести к крупным последствиям в плане безопасности, если вы применяете строковые литералы для хранения паролей, номеров кредитных карт или другой конфиденциальной информации.

У вас может возникнуть вопрос о том, каким образом задействовать такую информацию в разрабатываемых приложениях (в лучшем сценарии) или зачем вообще заботиться о метаданных (в худшем сценарии). Чтобы получить ответ, необходимо ознакомиться со службами рефлексии .NET Core. Следует отметить, что полезность рассматриваемых далее тем может стать ясной только ближе к концу главы, а потому наберитесь терпения.

На заметку! В разделе `METAINFO` вы также найдете несколько маркеров `CustomAttribute`, которые документируют атрибуты, применяемые внутри кодовой базы. Роль атрибутов .NET Core обсуждается позже в главе.

Понятие рефлексии

В мире .NET Core *рефлексией* называется процесс обнаружения типов во время выполнения. Службы рефлексии дают возможность получать программно ту же са-

мую информацию о метаданных, которую генерирует утилита `ildasm.exe`, используя дружественную объектную модель. Например, посредством рефлексии можно извлечь список всех типов, содержащихся внутри заданной сборки `*.dll` или `*.exe`, в том числе методы, поля, свойства и события, которые определены конкретным типом. Можно также динамически получать набор интерфейсов, поддерживаемых заданным типом, параметры метода и другие относящиеся к ним детали (базовые классы, пространства имен, данные манифеста и т.д.).

Как и любое другое пространство имен, `System.Reflection` (из сборки `System.Runtime.dll`) содержит набор связанных типов. В табл. 17.1 описаны основные члены `System.Reflection`, которые необходимо знать.

Таблица 17.1. Избранные члены пространства имен `System.Reflection`

Тип	Описание
`Assembly`	Этот абстрактный класс содержит несколько членов, которые позволяют загружать, исследовать и манипулировать сборкой
`AssemblyName`	Этот класс позволяет выяснить многочисленные детали, связанные с идентичностью сборки (информация о версии, информация о культуре и т.д.)
`EventInfo`	Этот абстрактный класс хранит информацию о заданном событии
`FieldInfo`	Этот абстрактный класс хранит информацию о заданном поле
`MemberInfo`	Этот абстрактный базовый класс определяет общее поведение для типов `EventInfo`, `FieldInfo`, `MethodInfo` и `PropertyInfo`
`MethodInfo`	Этот абстрактный класс содержит информацию о заданном методе
`Module`	Этот абстрактный класс позволяет получить доступ к заданному модулю внутри многофайловой сборки
`ParameterInfo`	Этот класс хранит информацию о заданном параметре
`PropertyInfo`	Этот абстрактный класс хранит информацию о заданном свойстве

Чтобы понять, каким образом задействовать пространство имен `System.Reflection` для программного чтения метаданных .NET Core, сначала следует ознакомиться с классом `System.Type`.

Класс `System.Type`

В классе `System.Type` определены члены, которые могут применяться для исследования метаданных типа, большое количество которых возвращают типы из пространства имен `System.Reflection`. Например, метод `Type.GetMethods()` возвращает массив объектов `MethodInfo`, метод `Type.GetFields()` — массив объектов `FieldInfo` и т.д. Полный перечень членов, доступных в `System.Type`, довольно велик, но в табл. 17.2 приведен список избранных членов, поддерживаемых `System.Type` (за исчерпывающими сведениями обращайтесь в документацию по .NET Core).

Таблица 17.2. Избранные члены `System.Type`

Член	Описание

`IsAbstract`
`IsArray`
`IsClass`
`IsCOMObject`
`IsEnum`
`IsGenericTypeDefinition`
`IsGenericParameter`
`IsInterface`
`IsPrimitive`
`IsNestedPrivate`
`IsNestedPublic`
`IsSealed`
`IsValueType`
	Эти свойства позволяют выяснять базовые характеристики типа, на который осуществляется ссылка (например, является ли он абстрактной сущностью, массивом, вложенным классом и т.п.)

`GetConstructors()`
`GetEvents()`
`GetFields()`
`GetInterfaces()`
`GetMembers()`
`GetMethods()`
`GetNestedTypes()`
`GetProperties()`
	Эти методы позволяют получать массив интересующих элементов (интерфейсов, методов, свойств и т.д.). Каждый метод возвращает связанный массив (скажем, `GetFields()` возвращает массив объектов `FieldInfo`, метод `GetMethods()` — массив объектов `MethodInfo` и т.д.). Следует отметить, что для каждого метода предусмотрена версия с именем в единственном числе (например, `GetMethod()`, `GetProperty()` и т.п.), которая позволяет извлекать специфический элемент по имени, а не массив всех связанных элементов

`FindMembers()`	Этот метод возвращает массив объектов `MemberInfo` на основе указанного критерия поиска
`GetType()`	Этот статический метод возвращает экземпляр `Type` с заданным строковым именем
`InvokeMember()`	Этот метод позволяет выполнять "позднее связывание" для указанного элемента. Вы узнаете о позднем связывании далее в главе

Получение информации о типе с помощью `System.Object.GetType()`

Экземпляр класса `Type` можно получать разнообразными способами. Тем не менее, есть одна вещь, которую делать невозможно — создавать объект `Type` напрямую, используя ключевое слово `new`, т.к. `Type` является абстрактным классом. Касательно первого способа вспомните, что в классе `System.Object` определен метод `GetType()`, который возвращает экземпляр класса `Type`, представляющий метаданные текущего объекта:

```
// Получить информацию о типе с применением экземпляра SportsCar.
SportsCar sc = new SportsCar();
Type t = sc.GetType();
```

Очевидно, что такой подход будет работать, только если подвергаемый рефлексии тип (`SportsCar` в данном случае) известен на этапе компиляции и в памяти присут-

ствует его экземпляр. С учетом этого ограничения должно быть понятно, почему инструменты вроде ildasm.exe не получают информацию о типе, непосредственно вызывая метод System.Object.GetType() для каждого типа — ведь утилита ildasm.exe не компилировалась вместе с вашими специальными сборками.

Получение информации о типе с помощью typeof()

Следующий способ получения информации о типе предполагает применение операции typeof:

```
// Получить информацию о типе с использованием операции typeof.
Type t = typeof(SportsCar);
```

В отличие от метода System.Object.GetType() операция typeof удобна тем, что она не требует предварительного создания экземпляра объекта перед получением информации о типе. Однако кодовой базе по-прежнему должно быть известно об исследуемом типе на этапе компиляции, поскольку typeof ожидает получения строго типизированного имени типа.

Получение информации о типе с помощью System.Type.GetType()

Для получения информации о типе в более гибкой манере можно вызывать статический метод GetType() класса System.Type и указывать полностью заданное строковое имя типа, который планируется изучить. При таком подходе знать тип, из которого будут извлекаться метаданные, на этапе компиляции *не* нужно, т.к. метод Type.GetType() принимает в качестве параметра экземпляр вездесущего класса System.String.

На заметку! Когда речь идет о том, что при вызове метода Type.GetType() знание типа на этапе компиляции не требуется, имеется в виду тот факт, что данный метод может принимать любое строковое значение (а не строго типизированную переменную). Разумеется, знать имя типа в строковом формате по-прежнему необходимо!

Метод Type.GetType() перегружен, позволяя указывать два булевских параметра, из которых один управляет тем, должно ли генерироваться исключение, если тип не удается найти, а второй отвечает за то, должен ли учитываться регистр символов в строке. В целях иллюстрации рассмотрим следующий код:

```
// Получить информацию о типе с использованием статического
// метода Type.GetType().
// (Не генерировать исключение, если тип SportsCar не удается найти,
// и игнорировать регистр символов.)
Type t = Type.GetType("CarLibrary.SportsCar", false, true);
```

В приведенном выше примере обратите внимание на то, что в строке, передаваемой методу GetType(), никак не упоминается сборка, внутри которой содержится интересующий тип. В этом случае делается предположение о том, что тип определен внутри сборки, выполняющейся в текущий момент. Тем не менее, когда необходимо получить метаданные для типа из внешней сборки, строковый параметр форматируется с использованием полностью заданного имени типа, за которым следует запятая и дружественное имя сборки (имя сборки без информации о версии), содержащей интересующий тип:

```
// Получить информацию о типе из внешней сборки.
Type t = Type.GetType("CarLibrary.SportsCar, CarLibrary");
```

Кроме того, в передаваемой методу `GetType()` строке может быть указан символ "плюс" (+) для обозначения *вложенного типа*. Пусть необходимо получить информацию о типе перечисления (`SpyOptions`), вложенного в класс по имени `JamesBondCar`. В таком случае можно написать следующий код:

```
// Получить информацию о типе для вложенного перечисления
// внутри текущей сборки.
Type t = Type.GetType("CarLibrary.JamesBondCar+SpyOptions");
```

Построение специального средства для просмотра метаданных

Чтобы ознакомиться с базовым процессом рефлексии (и выяснить полезность класса `System.Type`), создайте новый проект консольного приложения по имени `MyTypeViewer`. Приложение будет отображать детали методов, свойств, полей и поддерживаемых интерфейсов (в дополнение к другим интересным данным) для любого типа внутри `System.Runtime.dll` (вспомните, что все приложения .NET Core автоматически получают доступ к этой основной библиотеке классов платформы) или типа внутри самого приложения `MyTypeViewer`. После создания приложения не забудьте импортировать пространства имен `System`, `System.Reflection` и `System.Linq`:

```
// Эти пространства имен должны импортироваться для выполнения
// любой рефлексии!
using System;
using System.Linq;
using System.Reflection;
```

Рефлексия методов

В класс `Program` будут добавлены статические методы, каждый из которых принимает единственный параметр `System.Type` и возвращает `void`. Первым делом определите метод `ListMethods()`, который выводит имена методов, определенных во входном типе. Обратите внимание, что `Type.GetMethods()` возвращает массив объектов `System.Reflection.MethodInfo`, по которому можно осуществлять проход с помощью стандартного цикла `foreach`:

```
// Отобразить имена методов в типе.
static void ListMethods(Type t)
{
  Console.WriteLine("***** Methods *****");
  MethodInfo[] mi = t.GetMethods();
  foreach(MethodInfo m in mi)
  {
    Console.WriteLine("->{0}", m.Name);
  }
  Console.WriteLine();
}
```

Здесь просто выводится имя метода с применением свойства `MethodInfo.Name`. Как не трудно догадаться, класс `MethodInfo` имеет множество дополнительных членов, которые позволяют выяснить, является ли метод статическим, виртуальным,

обобщенным или абстрактным. Вдобавок тип `MethodInfo` дает возможность получить информацию о возвращаемом значении и наборе параметров метода. Чуть позже реализация `ListMethods()` будет немного улучшена.

При желании для перечисления имен методов можно было бы также построить подходящий запрос LINQ. Вспомните из главы 13, что технология LINQ to Object позволяет создавать строго типизированные запросы и применять их к коллекциям объектов в памяти. В качестве эмпирического правила запомните, что при обнаружении блоков с программной логикой циклов или принятия решений можно использовать соответствующий запрос LINQ. Скажем, предыдущий метод можно было бы переписать так, задействовав LINQ:

```
using System.Linq;
static void ListMethods(Type t)
{
  Console.WriteLine("***** Methods *****");
  var methodNames = from n in t.GetMethods() select n.Name;
  foreach (var name in methodNames)
  {
    Console.WriteLine("->{0}", name);
  }
  Console.WriteLine();
}
```

Рефлексия полей и свойств

Реализация метода `ListFields()` похожа. Единственным заметным отличием является вызов `Type.GetFields()` и результирующий массив элементов `FieldInfo`. И снова для простоты выводятся только имена каждого поля с применением запроса LINQ:

```
// Отобразить имена полей в типе.
static void ListFields(Type t)
{
  Console.WriteLine("***** Fields *****");
  var fieldNames = from f in t.GetFields() select f.Name;
  foreach (var name in fieldNames)
  {
    Console.WriteLine("->{0}", name);
  }
  Console.WriteLine();
}
```

Логика для отображения имен свойств типа аналогична:

```
// Отобразить имена свойств в типе.
static void ListProps(Type t)
{
  Console.WriteLine("***** Properties *****");
  var propNames = from p in t.GetProperties() select p.Name;
  foreach (var name in propNames)
  {
    Console.WriteLine("->{0}", name);
  }
  Console.WriteLine();
}
```

Рефлексия реализованных интерфейсов

Следующим создается метод по имени `ListInterfaces()`, который будет выводить имена любых интерфейсов, поддерживаемых входным типом. Один интересный момент здесь заключается в том, что вызов `GetInterfaces()` возвращает массив объектов `System.Type`! Это вполне логично, поскольку интерфейсы действительно являются типами:

```
// Отобразить имена интерфейсов, которые реализует тип.
static void ListInterfaces(Type t)
{
  Console.WriteLine("***** Interfaces *****");
  var ifaces = from i in t.GetInterfaces() select i;
  foreach(Type i in ifaces)
  {
    Console.WriteLine("->{0}", i.Name);
  }
}
```

На заметку! Имейте в виду, что большинство методов "получения" в `System.Type` (`GetMethods()`, `GetInterfaces()` и т.д.) перегружены, чтобы позволить указывать значения из перечисления `BindingFlags`. В итоге появляется высокий уровень контроля над тем, что в точности необходимо искать (например, только статические члены, только открытые члены, включать закрытые члены и т.д.). За более подробной информацией обращайтесь в документацию.

Отображение разнообразных дополнительных деталей

В качестве последнего, но не менее важного действия, осталось реализовать финальный вспомогательный метод, который будет отображать различные статистические данные о входном типе (является ли он обобщенным, какой его базовый класс, запечатан ли он и т.п.):

```
// Просто ради полноты картины.
static void ListVariousStats(Type t)
{
  Console.WriteLine("***** Various Statistics *****");

  Console.WriteLine("Base class is: {0}", t.BaseType);
                    // Базовый класс

  Console.WriteLine("Is type abstract? {0}", t.IsAbstract);
                    // Абстрактный?

  Console.WriteLine("Is type sealed? {0}", t.IsSealed);
                    // Запечатанный?

  Console.WriteLine("Is type generic? {0}", t.IsGenericTypeDefinition);
                    // Обобщенный?

  Console.WriteLine("Is type a class type? {0}", t.IsClass);
                    // Тип класса?
  Console.WriteLine();
}
```

Добавление операторов верхнего уровня

Операторы верхнего уровня в файле `Program.cs` запрашивают у пользователя полностью заданное имя типа. После получения этих строковых данных они передаются методу `Type.GetType()`, а результирующий объект `System.Type` отправляется каждому вспомогательному методу. Процесс повторяется до тех пор, пока пользователь не введет `Q` для прекращения работы приложения.

```
Console.WriteLine("***** Welcome to MyTypeViewer *****");
string typeName = "";

do
{
  Console.WriteLine("\nEnter a type name to evaluate");
                  // Пригласить ввести имя типа
  Console.Write("or enter Q to quit: ");       // или Q для завершения.

  // Получить имя типа.
  typeName = Console.ReadLine();

  // Пользователь желает завершить программу?
  if (typeName.Equals("Q",StringComparison.OrdinalIgnoreCase))
  {
    break;
  }

  // Попробовать отобразить информацию о типе.
  try
  {
    Type t = Type.GetType(typeName);
    Console.WriteLine("");
    ListVariousStats(t);
    ListFields(t);
    ListProps(t);
    ListMethods(t);
    ListInterfaces(t);
  }
  catch
  {
    Console.WriteLine("Sorry, can't find type"); // Не удается найти тип.
  }
} while (true);
```

В настоящий момент приложение `MyTypeViewer.exe` готово к тестовому запуску. Запустите его и введите следующие полностью заданные имена (не забывая, что `Type.GetType()` требует строковых имен с учетом регистра):

- `System.Int32`
- `System.Collections.ArrayList`
- `System.Threading.Thread`
- `System.Void`
- `System.IO.BinaryWriter`
- `System.Math`
- `MyTypeViewer.Program`

Ниже показан частичный вывод при указании `System.Math`:

```
***** Welcome to MyTypeViewer *****

Enter a type name to evaluate
or enter Q to quit: System.Math

***** Various Statistics *****
Base class is: System.Object
Is type abstract? True
Is type sealed? True
Is type generic? False
Is type a class type? True

***** Fields *****
->PI
->E

***** Properties *****

***** Methods *****
->Acos
->Asin
->Atan
->Atan2
->Ceiling
->Cos
...
```

Рефлексия статических типов

Если вы введете `System.Console` для предыдущего метода, тогда в первом вспомогательном методе сгенерируется исключение, потому что значением `t` будет `null`. Статические типы не могут загружаться с помощью метода `Type.GetType(typeName)`. Взамен придется использовать другой механизм — функцию `typeof` из `System.Type`. Модифицируйте программу для обработки особого случая `System.Console`:

```
Type t = Type.GetType(typeName);
if (t == null && typeName.Equals("System.Console",
   StringComparison.OrdinalIgnoreCase))
{
  t = typeof(System.Console);
}
```

Рефлексия обобщенных типов

При вызове `Type.GetType()` для получения описаний метаданных обобщенных типов должен использоваться специальный синтаксис, включающий символ обратной одинарной кавычки (`), за которым следует числовое значение, представляющее количество поддерживаемых параметров типа. Например, чтобы вывести описание метаданных `System.Collections.Generic.List<T>`, приложению потребуется передать следующую строку:

```
System.Collections.Generic.List`1
```

Здесь указано числовое значение 1, т.к. `List<T>` имеет только один параметр типа. Однако для применения рефлексии к типу `Dictionary<TKey, TValue>` понадобится предоставить значение 2:

```
System.Collections.Generic.Dictionary`2
```

Добавление операторов верхнего уровня

Операторы верхнего уровня в файле `Program.cs` запрашивают у пользователя полностью заданное имя типа. После получения этих строковых данных они передаются методу `Type.GetType()`, а результирующий объект `System.Type` отправляется каждому вспомогательному методу. Процесс повторяется до тех пор, пока пользователь не введет `Q` для прекращения работы приложения.

```
Console.WriteLine("***** Welcome to MyTypeViewer *****");
string typeName = "";

do
{
  Console.WriteLine("\nEnter a type name to evaluate");
                  // Пригласить ввести имя типа
  Console.Write("or enter Q to quit: ");       // или Q для завершения.

  // Получить имя типа.
  typeName = Console.ReadLine();

  // Пользователь желает завершить программу?
  if (typeName.Equals("Q",StringComparison.OrdinalIgnoreCase))
  {
    break;
  }

  // Попробовать отобразить информацию о типе.
  try
  {
    Type t = Type.GetType(typeName);
    Console.WriteLine("");
    ListVariousStats(t);
    ListFields(t);
    ListProps(t);
    ListMethods(t);
    ListInterfaces(t);
  }
  catch
  {
    Console.WriteLine("Sorry, can't find type"); // Не удается найти тип.
  }
} while (true);
```

В настоящий момент приложение `MyTypeViewer.exe` готово к тестовому запуску. Запустите его и введите следующие полностью заданные имена (не забывая, что `Type.GetType()` требует строковых имен с учетом регистра):

- `System.Int32`
- `System.Collections.ArrayList`
- `System.Threading.Thread`
- `System.Void`
- `System.IO.BinaryWriter`
- `System.Math`
- `MyTypeViewer.Program`

Ниже показан частичный вывод при указании `System.Math`:

```
***** Welcome to MyTypeViewer *****

Enter a type name to evaluate
or enter Q to quit: System.Math

***** Various Statistics *****
Base class is: System.Object
Is type abstract? True
Is type sealed? True
Is type generic? False
Is type a class type? True

***** Fields *****
->PI
->E

***** Properties *****

***** Methods *****
->Acos
->Asin
->Atan
->Atan2
->Ceiling
->Cos
...
```

Рефлексия статических типов

Если вы введете `System.Console` для предыдущего метода, тогда в первом вспомогательном методе сгенерируется исключение, потому что значением `t` будет `null`. Статические типы не могут загружаться с помощью метода `Type.GetType(typeName)`. Взамен придется использовать другой механизм — функцию `typeof` из `System.Type`. Модифицируйте программу для обработки особого случая `System.Console`:

```
Type t = Type.GetType(typeName);
if (t == null && typeName.Equals("System.Console",
    StringComparison.OrdinalIgnoreCase))
{
  t = typeof(System.Console);
}
```

Рефлексия обобщенных типов

При вызове `Type.GetType()` для получения описаний метаданных обобщенных типов должен использоваться специальный синтаксис, включающий символ обратной одинарной кавычки (`), за которым следует числовое значение, представляющее количество поддерживаемых параметров типа. Например, чтобы вывести описание метаданных `System.Collections.Generic.List<T>`, приложению потребуется передать следующую строку:

```
System.Collections.Generic.List`1
```

Здесь указано числовое значение 1, т.к. `List<T>` имеет только один параметр типа. Однако для применения рефлексии к типу `Dictionary<TKey, TValue>` понадобится предоставить значение 2:

```
System.Collections.Generic.Dictionary`2
```

```
    Console.WriteLine("\nEnter an assembly to evaluate");
                    // Пригласить ввести имя сборки
    Console.Write("or enter Q to quit: ");
                // или Q для завершения.
    // Получить имя сборки.
    asmName = Console.ReadLine();
    // Пользователь желает завершить программу?
    if (asmName.Equals("Q",StringComparison.OrdinalIgnoreCase))
    {
      break;
    }
    // Попробовать загрузить сборку.
    try
    {
      asm = Assembly.LoadFrom(asmName);
      DisplayTypesInAsm(asm);
    }
    catch
    {
      Console.WriteLine("Sorry, can't find assembly.");
                    // Сборка не найдена.
    }
  } while (true);
  static void DisplayTypesInAsm(Assembly asm)
  {
    Console.WriteLine("\n***** Types in Assembly *****");
    Console.WriteLine("->{0}", asm.FullName);
    Type[] types = asm.GetTypes();
    foreach (Type t in types)
    {
      Console.WriteLine("Type: {0}", t);
    }
    Console.WriteLine("");
  }
```

Если вы хотите проводить рефлексию по `CarLibrary.dll`, тогда перед запуском приложения `ExternalAssemblyReflector` понадобится скопировать двоичный файл `CarLibrary.dll` (из предыдущей главы) в каталог проекта (в случае применения Visual Studio Code) или в каталог `\bin\Debug\net5.0` самого приложения (в случае использования Visual Studio). После выдачи запроса введите `CarLibrary` (расширение необязательно); вывод будет выглядеть примерно так:

```
***** External Assembly Viewer *****
Enter an assembly to evaluate
or enter Q to quit: CarLibrary

***** Types in Assembly *****
->CarLibrary, Version=1.0.0.1, Culture=neutral, PublicKeyToken=null
Type: CarLibrary.MyInternalClass
Type: CarLibrary.EngineStateEnum
Type: CarLibrary.MusicMedia
Type: CarLibrary.Car
Type: CarLibrary.MiniVan
Type: CarLibrary.SportsCar
```

Метод `LoadFrom()` также может принимать абсолютный путь к файлу сборки, которую нужно просмотреть (скажем, `C:\MyApp\MyAsm.dll`). Благодаря этому методу вы можете передавать полный путь в своем проекте консольного приложения. Таким образом, если файл `CarLibrary.dll` находится в каталоге `C:\MyCode`, тогда вы можете ввести `C:\MyCode\CarLibrary` (обратите внимание, что расширение необязательно).

Рефлексия сборок инфраструктуры

Метод `Assembly.Load()` имеет несколько перегруженных версий. Одна из них разрешает указывать значение культуры (для локализованных сборок), а также номер версии и значение маркера открытого ключа (для сборок инфраструктуры). Коллективно многочисленные элементы, идентифицирующие сборку, называются *отображаемым именем*. Форматом отображаемого имени является строка пар "имя-значение", разделенных запятыми, которая начинается с дружественного имени сборки, а за ним следуют необязательные квалификаторы (в любом порядке). Вот как выглядит шаблон (необязательные элементы указаны в круглых скобках):

```
Имя (,Version = <старший номер>.<младший номер>.<номер сборки>.<номер редакции>)
(,Culture = <маркер культуры>) (,PublicKeyToken = <маркер открытого ключа>)
```

При создании отображаемого имени соглашение `PublicKeyToken=null` отражает тот факт, что требуется связывание и сопоставление со сборкой, не имеющей строгого имени. Вдобавок `Culture=""` указывает, что сопоставление должно осуществляться со стандартной культурой целевой машины. Вот пример:

```
//Загрузить версию 1.0.0.0 сборки CarLibrary, используя стандартную культуру
Assembly a = Assembly.Load(
   "CarLibrary, Version=1.0.0.0, PublicKeyToken=null, Culture=\"\"");
// В C# кавычки должны быть отменены с помощью символа обратной косой черты
```

Кроме того, следует иметь в виду, что пространство имен `System.Reflection` предлагает тип `AssemblyName`, который позволяет представлять показанную выше строковую информацию в удобной объектной переменной. Обычно класс `AssemblyName` применяется вместе с классом `System.Version`, который представляет собой объектно-ориентированную оболочку для номера версии сборки. После создания отображаемого имени его затем можно передавать перегруженной версии метода `Assembly.Load()`:

```
// Применение типа AssemblyName для определения отображаемого имени.
AssemblyName asmName;
asmName = new AssemblyName();
asmName.Name = "CarLibrary";
Version v = new Version("1.0.0.0");
asmName.Version = v;
Assembly a = Assembly.Load(asmName);
```

Чтобы загрузить сборку .NET Framework (не .NET Core), в параметре `Assembly.Load()` должно быть указано значение `PublicKeyToken`. В .NET Core это не требуется из-за того, что назначение строгих имен используется реже. Например, создайте новый проект консольного приложения по имени `FrameworkAssemblyViewer`, имеющий ссылку на пакет `Microsoft.EntityFrameworkCore`. Как вам уже известно, это можно сделать в интерфейсе командной строки .NET 5 (CLI):

```
dotnet new console -lang c# -n FrameworkAssemblyViewer
                   -o .\FrameworkAssemblyViewer -f net5.0
dotnet sln .\Chapter17_AllProjects.sln add .\FrameworkAssemblyViewer
dotnet add .\FrameworkAssemblyViewer
    package Microsoft.EntityFrameworkCore -v 5.0.0
```

Вспомните, что в случае ссылки на другую сборку копия этой сборки помещается в выходной каталог ссылаемого проекта. Скомпилируйте проект с применением CLI:

```
dotnet build
```

После создания проекта, добавления ссылки на `EntityFrameworkCode` и компиляции проекта сборку теперь можно загрузить и инспектировать. Поскольку количество типов в данной сборке довольно велико, приложение будет выводить только имена открытых перечислений, используя простой запрос LINQ:

```
using System;
using System.Linq;
using System.Reflection;

Console.WriteLine("***** The Framework Assembly Reflector App *****\n");

// Загрузить Microsoft.EntityFrameworkCore.dll.
var displayName =
  "Microsoft.EntityFrameworkCore, Version=5.0.0.0, Culture=\"\",
PublicKeyToken=adb9793829ddae60";
Assembly asm = Assembly.Load(displayName);
DisplayInfo(asm);
Console.WriteLine("Done!");  // Готово!
Console.ReadLine();

private static void DisplayInfo(Assembly a)
{
  Console.WriteLine("***** Info about Assembly *****");
  Console.WriteLine($"Asm Name: {a.GetName().Name}" );
                    // Имя сборки
  Console.WriteLine($"Asm Version: {a.GetName().Version}");
                    // Версия сборки
  Console.WriteLine($"Asm Culture: {a.GetName().CultureInfo.DisplayName}");
                    // Культура сборки
  Console.WriteLine("\nHere are the public enums:");
                    // Список открытых перечислений
  // Использовать запрос LINQ для нахождения открытых перечислений.
  Type[] types = a.GetTypes();
  var publicEnums =
    from pe in types
    where pe.IsEnum && pe.IsPublic
    select pe;
  foreach (var pe in publicEnums)
  {
    Console.WriteLine(pe);
  }
}
```

К настоящему моменту вы должны уметь работать с некоторыми основными членами пространства имен `System.Reflection` для получения метаданных во время выполнения. Конечно, необходимость в самостоятельном построении специальных

браузеров объектов в повседневной практике вряд ли будет возникать часто. Однако не забывайте, что службы рефлексии являются основой для многих распространенных действий программирования, включая *позднее связывание*.

Понятие позднего связывания

Позднее связывание представляет собой прием, который позволяет создавать экземпляр заданного типа и обращаться к его членам во время выполнения без необходимости в жестком кодировании факта его существования на этапе компиляции. При построении приложения, в котором производится позднее связывание с типом из внешней сборки, нет причин устанавливать ссылку на эту сборку; следовательно, в манифесте вызывающего кода она прямо не указывается.

На первый взгляд значимость позднего связывания оценить нелегко. Действительно, если есть возможность выполнить "раннее связывание" с объектом (например, добавить ссылку на сборку и выделить память под экземпляр типа с помощью ключевого слова `new`), то именно так следует поступать. Причина в том, что ранее связывание позволяет выявлять ошибки на этапе компиляции, а не во время выполнения. Тем не менее, позднее связывание играет важную роль в любом расширяемом приложении, которое может строиться. Пример построения такого "расширяемого" приложения будет приведен в конце главы, в разделе "Построение расширяемого приложения", а пока займемся исследованием роли класса `Activator`.

Класс `System.Activator`

Класс `System.Activator` играет ключевую роль в процессе позднего связывания .NET Core. В приведенном далее примере интересен только метод `Activator.CreateInstance()`, который применяется для создания экземпляра типа через позднее связывание. Этот метод имеет несколько перегруженных версий, обеспечивая достаточно высокую гибкость. Самая простая версия метода `CreateInstance()` принимает действительный объект `Type`, описывающий сущность, которую необходимо разместить в памяти на лету.

Создайте новый проект консольного приложения по имени `LateBindingApp` и с помощью ключевого слова `using` импортируйте в него пространства имен `System.IO` и `System.Reflection`. Модифицируйте файл `Program.cs`, как показано ниже:

```
using System;
using System.IO;
using System.Reflection;
// Это приложение будет загружать внешнюю сборку и
// создавать объект, используя позднее связывание.
Console.WriteLine("***** Fun with Late Binding *****");
// Попробовать загрузить локальную копию CarLibrary.
Assembly a = null;
try
{
  a = Assembly.LoadFrom("CarLibrary");
}
catch(FileNotFoundException ex)
{
  Console.WriteLine(ex.Message);
  return;
}
```

```
if(a != null)
{
  CreateUsingLateBinding(a);
}
Console.ReadLine();

static void CreateUsingLateBinding(Assembly asm)
{
  try
  {
    // Получить метаданные для типа MiniVan.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // Создать экземпляр MiniVan на лету.
    object obj = Activator.CreateInstance(miniVan);
    Console.WriteLine("Created a {0} using late binding!", obj);
  }
  catch(Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Перед запуском нового приложения понадобится вручную скопировать файл `CarLibrary.dll` в каталог с файлом проекта (или в подкаталог `bin\Debug\net5.0`, если вы работаете в Visual Studio) данного приложения.

На заметку! Не добавляйте ссылку на `CarLibrary.dll` в этом примере! Вся суть позднего связывания заключается в попытке создания объекта, который не известен на этапе компиляции.

Обратите внимание, что метод `Activator.CreateInstance()` возвращает экземпляр `System.Object`, а не строго типизированный объект `MiniVan`. Следовательно, если применить к переменной `obj` операцию точки, то члены класса `MiniVan` не будут видны. На первый взгляд может показаться, что проблему удастся решить с помощью явного приведения:

```
// Привести к типу MiniVan, чтобы получить доступ к его членам?
// Нет! Компилятор сообщит об ошибке!
object obj = (MiniVan)Activator.CreateInstance(minivan);
```

Однако из-за того, что в приложение не была добавлена ссылка на сборку `CarLibrary.dll`, использовать ключевое слово `using` для импортирования пространства имен `CarLibrary` нельзя, а значит невозможно и указывать тип `MiniVan` в операции приведения! Не забывайте, что смысл позднего связывания — создание экземпляров типов, о которых на этапе компиляции ничего не известно. Учитывая сказанное, возникает вопрос: как вызывать методы объекта `MiniVan`, сохраненного в ссылке на `System.Object`? Ответ: конечно же, с помощью рефлексии.

Вызов методов без параметров

Предположим, что требуется вызвать метод `TurboBoost()` объекта `MiniVan`. Вспомните, что упомянутый метод переводит двигатель в нерабочее состояние и затем отображает окно с соответствующим сообщением. Первый шаг заключается в получе-

нии объекта MethodInfo для метода TurboBoost() посредством Type.GetMethod(). Имея результирующий объект MethodInfo, можно вызвать MiniVan.TurboBoost() с помощью метода Invoke(). Метод MethodInfo.Invoke() требует указания всех параметров, которые подлежат передаче методу, представленному объектом MethodInfo. Параметры задаются в виде массива объектов System.Object (т.к. они могут быть самыми разнообразными сущностями).

Поскольку метод TurboBoost() не принимает параметров, можно просто передать null (т.е. сообщить, что вызываемый метод не имеет параметров). Обновите метод CreateUsingLateBinding() следующим образом:

```
static void CreateUsingLateBinding(Assembly asm)
{
  try
  {
    // Получить метаданные для типа Minivan.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // Создать объект MiniVan на лету.
    object obj = Activator.CreateInstance(miniVan);
    Console.WriteLine($"Created a {obj} using late binding!");

    // Получить информацию о TurboBoost.
    MethodInfo mi = miniVan.GetMethod("TurboBoost");

    // Вызвать метод (null означает отсутствие параметров).
    mi.Invoke(obj, null);
  }
  catch(Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Теперь после запуска приложения вы увидите в окне консоли сообщение о том, что двигатель вышел из строя ("Eek! Your engine block exploded!").

Вызов методов с параметрами

Когда позднее связывание нужно применять для вызова метода, ожидающего параметры, аргументы потребуется упаковать в слабо типизированный массив object. В версии класса Car с радиоприемником был определен следующий метод:

```
public void TurnOnRadio(bool musicOn, MusicMediaEnum mm)
  => MessageBox.Show(musicOn ? $"Jamming {mm}" : "Quiet time...");
```

Метод TurnOnRadio() принимает два параметра: булевское значение, которое указывает, должна ли быть включена музыкальная система в автомобиле, и перечисление, представляющее тип музыкального проигрывателя. Вспомните, что это перечисление определено так:

```
public enum MusicMediaEnum
{
  musicCd,     // 0
  musicTape,   // 1
  musicRadio,  // 2
  musicMp3     // 3
}
```

Ниже приведен код нового метода класса `Program`, в котором вызывается `TurnOnRadio()`. Обратите внимание на использование внутренних числовых значений перечисления `MusicMediaEnum`:

```
static void InvokeMethodWithArgsUsingLateBinding(Assembly asm)
{
  try
  {
    // Получить описание метаданных для типа SportsCar.
    Type sport = asm.GetType("CarLibrary.SportsCar");

    // Создать объект типа SportsCar.
    object obj = Activator.CreateInstance(sport);

    // Вызвать метод TurnOnRadio() с аргументами.
    MethodInfo mi = sport.GetMethod("TurnOnRadio");
    mi.Invoke(obj, new object[] { true, 2 });
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

В идеале к настоящему времени вы уже видите отношения между рефлексией, динамической загрузкой и поздним связыванием. Естественно, помимо раскрытых здесь возможностей API-интерфейс рефлексии предлагает много дополнительных средств, но вы уже должны быть в хорошей форме, чтобы погрузиться в дальнейшее изучение.

Вас все еще может интересовать вопрос: *когда* описанные приемы должны применяться в разрабатываемых приложениях? Ответ прояснится ближе к концу главы, а пока мы займемся исследованием роли атрибутов .NET Core.

Роль атрибутов .NET

Как было показано в начале главы, одной из задач компилятора .NET Core является генерация описаний метаданных для всех определяемых типов и для типов, на которые имеются ссылки. Помимо стандартных метаданных, содержащихся в любой сборке, платформа .NET Core предоставляет программистам способ встраивания в сборку дополнительных метаданных с использованием *атрибутов*. Выражаясь кратко, атрибуты — это всего лишь аннотации кода, которые могут применяться к заданному типу (классу, интерфейсу, структуре и т.п.), члену (свойству, методу и т.д.), сборке или модулю.

Атрибуты .NET Core представляют собой типы классов, расширяющие абстрактный базовый класс `System.Attribute`. По мере изучения пространств имен .NET Core вы найдете много предопределенных атрибутов, которые можно использовать в своих приложениях. Более того, вы также можете строить собственные атрибуты для дополнительного уточнения поведения своих типов путем создания нового типа, производного от `Attribute`.

Библиотеки базовых классов .NET Core предлагают атрибуты в различных пространствах имен. В табл. 17.3 описаны некоторые (*безусловно*, далеко не все) предопределенные атрибуты.

Таблица 17.3. Небольшая выборка предопределенных атрибутов

Атрибут	Описание
`[CLSCompliant]`	Заставляет аннотированный элемент соответствовать правилам CLS (Common Language Specification — общеязыковая спецификация). Вспомните, что совместимые с CLS типы могут гарантированно использоваться во всех языках программирования .NET Core
`[DllImport]`	Позволяет коду .NET Core обращаться к любой неуправляемой библиотеке кода C или C++, включая API-интерфейс операционной системы
`[Obsolete]`	Помечает устаревший тип или член. Если другие программисты попытаются использовать такой элемент, то они получат соответствующее предупреждение от компилятора

Важно понимать, что когда вы применяете атрибуты в своем коде, встроенные метаданные по существу бесполезны до тех пор, пока другая часть программного обеспечения явно не запросит такую информацию посредством рефлексии. В противном случае метаданные, встроенные в сборку, игнорируются и не причиняют никакого вреда.

Потребители атрибутов

Как нетрудно догадаться, в состав .NET Core входят многочисленные утилиты, которые действительно ищут разнообразные атрибуты. Сам компилятор C# (`csc.exe`) запрограммирован на обнаружение различных атрибутов при проведении компиляции. Например, встретив атрибут `[CLSCompilant]`, компилятор автоматически проверяет помеченный им элемент и удостоверяется в том, что в нем открыт доступ только к конструкциям, совместимым с CLS. Еще один пример: если компилятор обнаруживает элемент с атрибутом `[Obsolete]`, тогда он отображает в окне Error List (Список ошибок) среды Visual Studio сообщение с предупреждением.

В дополнение к инструментам разработки многие методы в библиотеках базовых классов .NET Core изначально запрограммированы на распознавание определенных атрибутов посредством рефлексии. В главе 20 рассматривается сериализация XML и JSON, которая задействует атрибуты для управления процессом сериализации.

Наконец, можно строить приложения, способные распознавать специальные атрибуты, а также любые атрибуты из библиотек базовых классов .NET Core. По сути, тем самым создается набор "ключевых слов", которые понимает специфичное множество сборок.

Применение атрибутов в C#

Чтобы ознакомиться со способами применения атрибутов в C#, создайте новый проект консольного приложения по имени `ApplyingAttributes` и добавьте ссылку на `System.Text.Json`. Предположим, что необходимо построить класс под названием `Motorcycle` (мотоцикл), который может сохраняться в формате JSON. Если какое-то поле сохраняться не должно, тогда к нему следует применить атрибут `[JsonIgnore]`.

```
public class Motorcycle
{
  [JsonIgnore]
  public float weightOfCurrentPassengers;
```

```
  // Эти поля остаются сериализируемыми.
  public bool hasRadioSystem;
  public bool hasHeadSet;
  public bool hasSissyBar;
}
```

На заметку! Атрибут применяется только к элементу, находящемуся непосредственно после него.

В данный момент пусть вас не беспокоит фактический процесс сериализации объектов (он подробно рассматривается в главе 20). Просто знайте, что для применения атрибута его имя должно быть помещено в квадратные скобки.

Нетрудно догадаться, что к одиночному элементу можно применять множество атрибутов. Предположим, что у вас есть унаследованный тип класса C# (`HorseAndBuggy`), который был снабжен атрибутом, чтобы иметь специальное пространство имен XML. Со временем кодовая база изменилась, и класс теперь считается устаревшим для текущей разработки. Вместо того чтобы удалять определение такого класса из кодовой базы (с риском нарушения работы существующего программного обеспечения), его можно пометить атрибутом `[Obsolete]`. Для применения множества атрибутов к одному элементу просто используйте список с разделителями-запятыми:

```
using System;
using System.Xml.Serialization;

namespace ApplyingAttributes
{
  [XmlRoot(Namespace = "http://www.MyCompany.com"),
           Obsolete("Use another vehicle!")]
                    // Используйте другое транспортное средство!
  public class HorseAndBuggy
  {
    // ...
  }
}
```

В качестве альтернативы применить множество атрибутов к единственному элементу можно также, указывая их друг за другом (конечный результат будет идентичным):

```
[XmlRoot(Namespace = "http://www.MyCompany.com")]
[Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
  // ...
}
```

Сокращенная система обозначения атрибутов C#

Заглянув в документацию по .NET Core, вы можете заметить, что действительным именем класса, представляющего атрибут `[Obsolete]`, является `ObsoleteAttribute`, а не просто `Obsolete`. По соглашению имена всех атрибутов .NET Core (включая специальные атрибуты, которые создаете вы сами) снабжаются суффиксом `Attribute`. Тем не менее, чтобы упростить процесс применения атрибутов, в языке C# не требуется обязательный ввод суффикса `Attribute`. Учитывая это, показанная ниже вер-

сия типа HorseAndBuggy идентична предыдущей версии (но влечет за собой более объемный клавиатурный набор):

```
[SerializableAttribute]
[ObsoleteAttribute("Use another vehicle!")]
public class HorseAndBuggy
{
  // ...
}
```

Имейте в виду, что такая сокращенная система обозначения для атрибутов предлагается только в C#. Ее поддерживают не все языки .NET Core.

Указание параметров конструктора для атрибутов

Обратите внимание, что атрибут [Obsolete] может принимать то, что выглядит как параметр конструктора. Если вы просмотрите формальное определение атрибута [Obsolete], щелкнув на нем правой кнопкой мыши в окне кода и выбрав в контекстном меню пункт Go To Definition (Перейти к определению), то обнаружите, что данный класс на самом деле предоставляет конструктор, принимающий System.String:

```
public sealed class ObsoleteAttribute : Attribute
{
  public ObsoleteAttribute(string message, bool error);
  public ObsoleteAttribute(string message);
  public ObsoleteAttribute();
  public bool IsError { get; }
  public string? Message { get; }
}
```

Важно понимать, что когда вы снабжаете атрибут параметрами конструктора, этот атрибут *не* размещается в памяти до тех пор, пока к параметрам не будет применена рефлексия со стороны другого типа или внешнего инструмента. Строковые данные, определенные на уровне атрибутов, просто сохраняются внутри сборки в виде блока метаданных.

Атрибут [Obsolete] в действии

Теперь, поскольку класс HorseAndBuggy помечен как устаревший, следующая попытка выделения памяти под его экземпляр:

```
using System;
using ApplyingAttributes;

Console.WriteLine("Hello World!");
HorseAndBuggy mule = new HorseAndBuggy();
```

приводит к выдаче компилятором предупреждающего сообщения, а именно — предупреждения CS0618 с сообщением, включающим информацию, которая передавалась атрибуту:

```
'HorseAndBuggy' is obsolete: 'Use another vehicle!'
HorseAndBuggy устарел: Используйте другое транспортное средство!
```

Среды Visual Studio и Visual Studio Code оказывают помощь также посредством IntelliSense, получая информацию через рефлексию.

На рис. 17.1 показаны результаты действия атрибута `[Obsolete]` в Visual Studio, а на рис. 17.2 — в Visual Studio Code. Обратите внимание, что в обеих средах используется термин *deprecated* вместо *obsolete*.

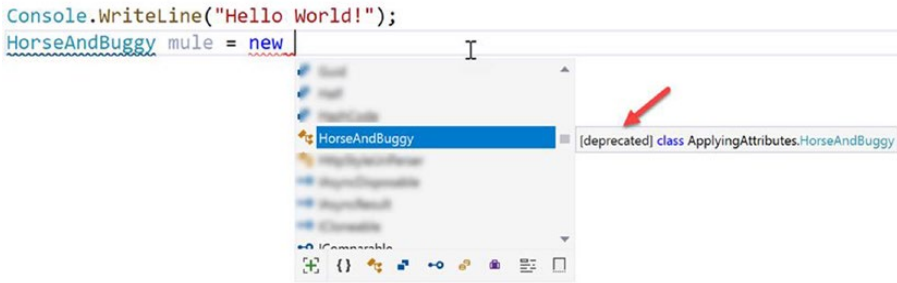

Рис. 17.1. Атрибуты в действии в среде Visual Studio

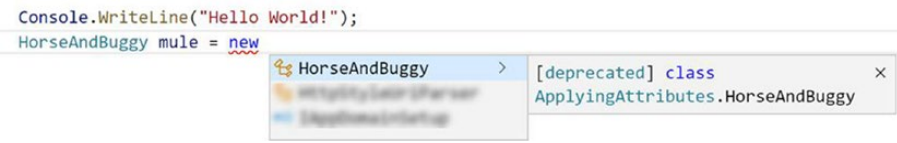

Рис. 17.2. Атрибуты в действии в среде Visual Studio Code

В идеальном случае к настоящему моменту вы уже должны понимать перечисленные ниже ключевые моменты, касающиеся атрибутов .NET Core:

- атрибуты представляют собой классы, производные от `System.Attribute`;
- атрибуты дают в результате встроенные метаданные;
- атрибуты в основном бесполезны до тех пор, пока другой агент не проведет в их отношении рефлексию;
- атрибуты в языке C# применяются с использованием квадратных скобок.

А теперь давайте посмотрим, как реализовывать собственные специальные атрибуты и создавать специальное программное обеспечение, которое выполняет рефлексию по встроенным метаданным.

Построение специальных атрибутов

Первый шаг при построении специального атрибута предусматривает создание нового класса, производного от `System.Attribute`. Не отклоняясь от автомобильной темы, повсеместно встречающейся в книге, создайте новый проект типа Class Library (Библиотека классов) на C# под названием `AttributedCarLibrary`. В этой сборке будет определено несколько классов для представления транспортных средств, каждый из которых описан с использованием специального атрибута по имени `VehicleDescriptionAttribute`:

```
using System;
// Специальный атрибут.
public sealed class VehicleDescriptionAttribute :Attribute
{
  public string Description { get; set; }

  public VehicleDescriptionAttribute(string description)
    => Description = description;
  public VehicleDescriptionAttribute(){ }
}
```

Как видите, класс `VehicleDescriptionAttribute` поддерживает фрагмент строковых данных, которым можно манипулировать с помощью автоматического свойства (`Description`). Помимо того факта, что данный класс является производным от `System.Attribute`, ничего примечательного в его определении нет.

На заметку! По причинам, связанным с безопасностью, установившейся практикой в .NET Core считается проектирование всех специальных атрибутов как запечатанных. На самом деле среды Visual Studio и Visual Studio Code предлагают фрагмент кода под названием `Attribute`, который позволяет сгенерировать в окне редактора кода новый класс, производный от `System.Attribute`. Для раскрытия любого фрагмента кода необходимо набрать его имя и нажать клавишу <Tab> (один раз в Visual Studio Code и два раза в Visual Studio).

Применение специальных атрибутов

С учетом того, что класс `VehicleDescriptionAttribute` является производным от `System.Attribute`, теперь можно аннотировать транспортные средства. В целях тестирования добавьте в новую библиотеку классов следующие файлы классов:

```
// Motorcycle.cs
namespace AttributedCarLibrary
{
  // Назначить описание с помощью "именованного свойства".
  [Serializable]
  [VehicleDescription(Description = "My rocking Harley")]
                                    // Мой покачивающийся Харли
  public class Motorcycle
  {
  }
}

// HorseAndBuggy.cs
namespace AttributedCarLibrary
{
  [Serializable]
  [Obsolete ("Use another vehicle!")]
  [VehicleDescription("The old gray mare, she ain't what she used to be...")]
                  // Старая серая лошадка, она уже не та...
  public class HorseAndBuggy
  {
  }
}
```

```
// Winnebago.cs
namespace AttributedCarLibrary
{
  [VehicleDescription("A very long, slow, but feature-rich auto")]
                  // Очень длинный, медленный, но обладающий высокими
                  // техническими характеристиками автомобиль
  public class Winnebago
  {
  }
}
```

Синтаксис именованных свойств

Обратите внимание, что классу `Motorcycle` назначается описание с использованием нового фрагмента синтаксиса, связанного с атрибутами, который называется *именованным свойством*. В конструкторе первого атрибута `[VehicleDescription]` лежащие в основе строковые данные устанавливаются с применением свойства `Description`. Когда внешний агент выполнит рефлексию для этого атрибута, свойству `Description` будет передано указанное значение (синтаксис именованных свойств разрешен, только если атрибут предоставляет поддерживающее запись свойство .NET Core).

По контрасту для типов `HorseAndBuggy` и `Winnebago` синтаксис именованных свойств не используется, а строковые данные просто передаются через специальный конструктор. В любом случае после компиляции сборки `AttributedCarLibrary` с помощью утилиты `ildasm.exe` можно просмотреть добавленные описания метаданных. Например, ниже показано встроенное описание класса `Winnebago`:

```
// CustomAttribute #1
// -------------------------------------------------------
//   CustomAttribute Type: 06000005
//   CustomAttributeName:
//   AttributedCarLibrary.VehicleDescriptionAttribute :: instance void
.ctor(class System.String)
//   Length: 45
//   Value : 01 00 28 41 20 76 65 72 79 20 6c 6f 6e 67 2c 20 >(A very long, <
//   : 73 6c 6f 77 2c 20 62 75 74 20 66 65 61 74 75 72 >slow, but feature<
//   : 65 2d 72 69 63 68 20 61 75 74 6f 00 00 >e-rich auto <
//   ctor args: ("A very long, slow, but feature-rich auto")
```

Ограничение использования атрибутов

По умолчанию специальные атрибуты могут быть применены практически к любому аспекту кода (методам, классам, свойствам и т.д.). Таким образом, если бы это имело смысл, то `VehicleDescription` можно было бы использовать для уточнения методов, свойств или полей (помимо прочего):

```
[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
  [VehicleDescription("My rocking CD player")]
  public void PlayMusic(bool On)
  {
    ...
  }
}
```

В одних случаях такое поведение является точно таким, какое требуется, но в других может возникнуть желание создать специальный атрибут, применяемый только к избранным элементам кода. Чтобы ограничить область действия специального атрибута, понадобится добавить к его определению атрибут [AttributeUsage], который позволяет предоставлять любую комбинацию значений (посредством операции "ИЛИ") из перечисления AttributeTargets:

```
// Это перечисление определяет возможные целевые элементы для атрибута.
public enum AttributeTargets
{
  All, Assembly, Class, Constructor,
  Delegate, Enum, Event, Field, GenericParameter,
  Interface, Method, Module, Parameter,
  Property, ReturnValue, Struct
}
```

Кроме того, атрибут [AttributeUsage] допускает необязательную установку именованного свойства (AllowMultiple), которое указывает, может ли атрибут применяться к тому же самому элементу более одного раза (стандартным значением является false). Вдобавок [AttributeUsage] разрешает указывать, должен ли атрибут наследоваться производными классами, с использованием именованного свойства Inherited (со стандартным значением true).

Модифицируйте определение VehicleDescriptionAttribute для указания на то, что атрибут [VehicleDescription] может применяться только к классу или структуре:

```
// На этот раз для аннотирования специального атрибута
// используется атрибут AttributeUsage.
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
                Inherited = false)]
public sealed class VehicleDescriptionAttribute : System.Attribute
{
  ...
}
```

Теперь если разработчик попытается применить атрибут [VehicleDescription] не к классу или структуре, то компилятор сообщит об ошибке.

Атрибуты уровня сборки

Атрибуты можно также применять ко всем типам внутри отдельной сборки, используя дескриптор [assembly:]. Например, предположим, что необходимо обеспечить совместимость с CLS для всех открытых членов во всех открытых типах, определенных внутри сборки. Рекомендуется добавить в проект новый файл по имени AssemblyAttributes.cs (не AssemblyInfo.cs, т.к. он генерируется автоматически) и поместить в него атрибуты уровня сборки.

На заметку! Какая-либо формальная причина для использования отдельного файла отсутствует; это связано чисто с удобством поддержки вашего кода. Помещение атрибутов сборки в отдельный файл проясняет тот факт, что в вашем проекте используются атрибуты уровня сборки, и показывает, где они находятся.

При добавлении в проект атрибутов уровня сборки или модуля имеет смысл придерживаться следующей рекомендуемой схемы для файла кода:

```
// Первыми перечислить операторы using.
using System;

// Теперь перечислить атрибуты уровня сборки или модуля.
// Обеспечить совместимость с CLS для всех открытых типов в данной сборке.
[assembly: CLSCompliant(true)]
```

Если теперь добавить фрагмент кода, выходящий за рамки спецификации CLS (вроде открытого элемента данных без знака), тогда компилятор выдаст предупреждение:

```
// Тип ulong не соответствует спецификации CLS.
public class Winnebago
{
  public ulong notCompliant;
}
```

На заметку! В .NET Core внесены два значительных изменения. Первое касается того, что файл `AssemblyInfo.cs` теперь генерируется автоматически из свойств проекта и настраивать его не рекомендуется. Второе (и связанное) изменение заключается в том, что многие из предшествующих атрибутов уровня сборки (`Version`, `Company` и т.д.) были заменены свойствами в файле проекта.

Использование файла проекта для атрибутов сборки

Как было демонстрировалось в главе 16 с классом `InternalsVisibleToAttribute`, атрибуты сборки можно также добавлять в файл проекта. Загвоздка здесь в том, что применять подобным образом можно только однострочные атрибуты параметров. Это справедливо для свойств, которые могут устанавливаться на вкладке Package (Пакет) в окне свойств проекта.

На заметку! На момент написания главы в хранилище GitHub для MSBuild шло активное обсуждение относительно добавления возможности поддержки нестроковых параметров, что позволило бы добавлять атрибут `CLSCompliant` с использованием файла проекта вместо файла `*.cs`.

Установите несколько свойств (таких как `Authors`, `Description`), щелкнув правой кнопкой мыши на имени проекта в окне Solution Explorer, выберите в контекстном меню пункт Properties (Свойства) и в открывшемся окне свойств перейдите на вкладку Package. Кроме того, добавьте `InternalsVisibleToAttribute`, как делалось в главе 16. Содержимое вашего файла проекта должно выглядеть примерно так, как представленное ниже:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <Authors>Philip Japikse</Authors>
    <Company>Apress</Company>
    <Description>This is a simple car library with attributes
    </Description>
  </PropertyGroup>
```

```
    <ItemGroup>
      <AssemblyAttribute Include=
         "System.Runtime.CompilerServices.InternalsVisibleToAttribute">
        <_Parameter1>CSharpCarClient</_Parameter1>
      </AssemblyAttribute>
    </ItemGroup>
  </Project>
```

После компиляции своего проекта перейдите в каталог `\obj\Debug\net5.0` и отыщите файл `AttributedCarLibrary.AssemblyInfo.cs`. Открыв его, вы увидите установленные свойства в виде атрибутов (к сожалению, они не особо читабельны в таком формате):

```
using System;
using System.Reflection;

[assembly: System.Runtime.CompilerServices.InternalsVisibleToAttribute
("CSharpCarClient")]
[assembly: System.Reflection.AssemblyCompanyAttribute("Philip Japikse")]
[assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
[assembly: System.Reflection.AssemblyDescriptionAttribute("This is a
sample car library with attributes")]
[assembly: System.Reflection.AssemblyFileVersionAttribute("1.0.0.0")]
[assembly: System.Reflection.AssemblyInformationalVersionAttribute("1.0.0")]
[assembly: System.Reflection.AssemblyProductAttribute("AttributedCarLibrary")]
[assembly: System.Reflection.AssemblyTitleAttribute("AttributedCarLibrary")]
[assembly: System.Reflection.AssemblyVersionAttribute("1.0.0.0")]
```

И последнее замечание, касающееся атрибутов сборки: вы можете отключить генерацию файла `AssemblyInfo.cs`, если хотите управлять процессом самостоятельно.

Рефлексия атрибутов с использованием раннего связывания

Вспомните, что атрибуты остаются бесполезными до тех пор, пока к их значениям не будет применена рефлексия в другой части программного обеспечения. После обнаружения атрибута другая часть кода может предпринять необходимый образ действий. Подобно любому приложению "другая часть программного обеспечения" может обнаруживать присутствие специального атрибута с использованием либо раннего, либо позднего связывания. Для применения раннего связывания определение интересующего атрибута (в данном случае `VehicleDescriptionAttribute`) должно находиться в клиентском приложении на этапе компиляции. Учитывая то, что специальный атрибут определен в сборке `AttributedCarLibrary` как открытый класс, раннее связывание будет наилучшим выбором.

Чтобы проиллюстрировать процесс рефлексии специальных атрибутов, вставьте в решение новый проект консольного приложения по имени `VehicleDescriptionAttributeReader`. Добавьте в него ссылку на проект `AttributedCarLibrary`. Выполните приведенные далее команды CLI (каждая должна вводиться по отдельности):

```
dotnet new console -lang c# -n VehicleDescriptionAttributeReader
        -o .\VehicleDescriptionAttributeReader -f net5.0
dotnet sln .\Chapter17_AllProjects.sln add .\VehicleDescriptionAttributeReader
dotnet add VehicleDescriptionAttributeReader reference .\AttributedCarLibrary
```

Поместите в файл Program.cs следующий код:

```
using System;
using AttributedCarLibrary;

Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
ReflectOnAttributesUsingEarlyBinding();
Console.ReadLine();
static void ReflectOnAttributesUsingEarlyBinding()
{
  // Получить объект Type, представляющий тип Winnebago.
  Type t = typeof(Winnebago);

  // Получить все атрибуты Winnebago.
  object[] customAtts = t.GetCustomAttributes(false);

  // Вывести описание.
  foreach (VehicleDescriptionAttribute v in customAtts)
  {
    Console.WriteLine("-> {0}\n", v.Description);
  }
}
```

Метод Type.GetCustomAttributes() возвращает массив объектов со всеми атрибутами, примененными к члену, который представлен объектом Type (булевский параметр управляет тем, должен ли поиск распространяться вверх по цепочке наследования). После получения списка атрибутов осуществляется проход по всем элементам VehicleDescriptionAttribute с отображением значения свойства Description.

Рефлексия атрибутов с использованием позднего связывания

В предыдущем примере для вывода описания транспортного средства типа Winnebago применялось ранее связывание. Это было возможно благодаря тому, что тип класса VehicleDescriptionAttribute определен в сборке AttributedCarLibrary как открытый член. Для рефлексии атрибутов также допускается использовать динамическую загрузку и позднее связывание.

Добавьте к решению новый проект консольного приложения по имени VehicleDescriptionAttributeReaderLateBinding, установите его в качестве стартового и скопируйте сборку AttributedCarLibrary.dll в каталог проекта (или в \bin\Debug\net5.0, если вы работаете в Visual Studio). Модифицируйте файл Program.cs, как показано ниже:

```
using System;
using System.Reflection;

Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
ReflectAttributesUsingLateBinding();
Console.ReadLine();

static void ReflectAttributesUsingLateBinding()
{
  try
  {
    // Загрузить локальную копию сборки AttributedCarLibrary.
    Assembly asm = Assembly.LoadFrom("AttributedCarLibrary");
```

```
    // Получить информацию о типе VehicleDescriptionAttribute.
    Type vehicleDesc =
      asm.GetType("AttributedCarLibrary.VehicleDescriptionAttribute");
    // Получить информацию о типе свойства Description.
    PropertyInfo propDesc = vehicleDesc.GetProperty("Description");
    // Получить все типы в сборке.
    Type[] types = asm.GetTypes();
    //Пройти по всем типам и получить любые атрибуты VehicleDescriptionAttribute
    foreach (Type t in types)
    {
      object[] objs = t.GetCustomAttributes(vehicleDesc, false);
      // Пройти по каждому VehicleDescriptionAttribute и вывести
      // описание, используя позднее связывание.
      foreach (object o in objs)
      {
        Console.WriteLine("-> {0}: {1}\n", t.Name,
          propDesc.GetValue(o, null));
      }
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Если вы прорабатывали примеры, рассмотренные ранее в главе, тогда приведенный код должен быть более или менее понятен. Единственный интересный момент здесь связан с применением метода `PropertyInfo.GetValue()`, который служит для активизации средства доступа к свойству. Вот как выглядит вывод, полученный в результате выполнения текущего примера:

```
***** Value of VehicleDescriptionAttribute *****
-> Motorcycle: My rocking Harley
-> HorseAndBuggy: The old gray mare, she ain't what she used to be...
-> Winnebago: A very long, slow, but feature-rich auto
```

Практическое использование рефлексии, позднего связывания и специальных атрибутов

Хотя вы видели многочисленные примеры применения этих приемов, вас по-прежнему может интересовать, когда использовать рефлексию, динамическое связывание и специальные атрибуты в своих программах. Действительно, данные темы могут показаться в большей степени относящимися к академической стороне программирования (что в зависимости от вашей точки зрения может быть как отрицательным, так и положительным аспектом). Для содействия в отображении указанных тем на реальные ситуации необходим более серьезный пример. Предположим, что вы работаете в составе команды программистов, которая занимается построением приложения, соблюдая требование о том, что продукт должен быть расширяемым за счет использования добавочных сторонних инструментов.

Что понимается под *расширяемостью*? Возьмем IDE-среду Visual Studio. Когда это приложение разрабатывалось, в его кодовую базу были вставлены многочисленные "привязки", чтобы позволить другим производителям программного обеспечения подключать специальные модули к IDE-среде. Очевидно, что у разработчиков Visual Studio отсутствовал какой-либо способ установки ссылок на внешние сборки .NET Core, которые на тот момент еще не были созданы (и потому раннее связывание недоступно), тогда как они обеспечили наличие в приложении необходимых привязок? Ниже представлен один из возможных способов решения задачи.

1. Во-первых, расширяемое приложение должно предоставлять некоторый механизм ввода, позволяющий пользователю указать модуль для подключения (наподобие диалогового окна или флага командной строки). Это требует *динамической загрузки*.
2. Во-вторых, расширяемое приложение должно иметь возможность выяснять, поддерживает ли модуль корректную функциональность (такую как набор обязательных интерфейсов), необходимую для его подключения к среде. Это требует *рефлексии*.
3. В-третьих, расширяемое приложение должно получать ссылку на требуемую инфраструктуру (вроде набора интерфейсных типов) и вызывать члены для запуска лежащей в основе функциональности. Это может требовать *позднего связывания*.

Попросту говоря, если расширяемое приложение изначально запрограммировано для запрашивания специфических интерфейсов, то во время выполнения оно в состоянии выяснять, может ли быть активизирован интересующий тип. После успешного прохождения такой проверки тип может поддерживать дополнительные интерфейсы, которые формируют полиморфную фабрику его функциональности. Именно этот подход был принят командой разработчиков Visual Studio, и вопреки тому, что вы могли подумать, в нем нет ничего сложного!

Построение расширяемого приложения

В последующих разделах будет рассмотрен пример создания расширяемого приложения, которое может быть дополнено функциональностью внешних сборок. Расширяемое приложение образовано из следующих сборок.

- `CommonSnappableTypes.dll`. Эта сборка содержит определения типов, которые будут использоваться каждым объектом оснастки. На нее будет напрямую ссылаться расширяемое приложение.
- `CSharpSnapIn.dll`. Оснастка, написанная на C#, в которой задействованы типы из сборки `CommonSnappableTypes.dll`.
- `VBSnapIn.dll`. Оснастка, написанная на Visual Basic, в которой применяются типы из сборки `CommonSnappableTypes.dll`.
- `MyExtendableApp.exe`. Консольное приложение, которое может быть расширено функциональностью каждой оснастки.

В приложении будут использоваться динамическая загрузка, рефлексия и позднее связывание для динамического получения функциональности сборок, о которых заранее ничего не известно.

На заметку! Вы можете подумать о том, что вам вряд ли будет ставиться задача построения консольного приложения, и тут вы вероятно правы! Бизнес-приложения, создаваемые на языке C#, обычно относятся к категории интеллектуальных клиентов (Windows Forms или WPF), веб-приложений/служб REST (ASP.NET Core) или автоматических процессов (функций Azure, служб Windows и т.д.). Консольные приложения применяются здесь, чтобы сосредоточиться на специфических концепциях примеров, в данном случае — на динамической загрузке, рефлексии и позднем связывании. Позже в книге вы узнаете, как строить "реальные" пользовательские приложения с использованием WPF и ASP.NET Core.

Построение мультипроектного решения `ExtendableApp`

Большинство приложений, созданных ранее в книге, были автономными проектами с небольшими исключениями (вроде предыдущего приложения). Так делалось для того, чтобы сохранять примеры простыми и четко ориентированными на демонстрируемые в них аспекты. Однако в реальном процессе разработки обычно приходится работать с множеством проектов в одном решении.

Создание решения и проектов с помощью интерфейса командной строки

Открыв окно интерфейса CLI, введите следующие команды, чтобы создать новое решение, проекты для библиотек классов и консольного приложения, а также ссылки на проекты:

```
dotnet new sln -n Chapter17_ExtendableApp

dotnet new classlib -lang c# -n CommonSnappableTypes
    -o .\CommonSnappableTypes -f net5.0
dotnet sln .\Chapter17_ExtendableApp.sln add .\CommonSnappableTypes

dotnet new classlib -lang c# -n CSharpSnapIn -o .\CSharpSnapIn -f net5.0
dotnet sln .\Chapter17_ExtendableApp.sln add .\CSharpSnapIn
dotnet add CSharpSnapin reference CommonSnappableTypes

dotnet new classlib -lang vb -n VBSnapIn -o .\VBSnapIn -f net5.0
dotnet sln .\Chapter17_ExtendableApp.sln add .\VBSnapIn
dotnet add VBSnapIn reference CommonSnappableTypes

dotnet new console -lang c# -n MyExtendableApp -o .\MyExtendableApp -f net5.0
dotnet sln .\Chapter17_ExtendableApp.sln add .\MyExtendableApp
dotnet add MyExtendableApp reference CommonSnappableTypes
```

Добавление событий `PostBuild` в файлы проектов

При компиляции проекта (либо в Visual Studio, либо в командной строке) существуют события, к которым можно привязываться. Например, после каждой успешной компиляции нужно копировать две сборки оснасток в каталог проекта консольного приложения (в случае отладки посредством `dotnet run`) и в выходной каталог консольного приложения (при отладке в Visual Studio). Для этого будут использоваться несколько встроенных макросов.

Вставьте в файлы `CSharpSnapIn.csproj` и `VBSnapIn.vbproj` приведенный ниже блок разметки, который копирует скомпилированную сборку в каталог проекта `MyExtendableApp` и в выходной каталог (`MyExtendableApp\bin\debug\net5.0`):

```
<Target Name="PostBuild" AfterTargets="PostBuildEvent">
  <Exec Command="copy $(TargetPath) $(SolutionDir)MyExtendableApp\
$(OutDir)$(TargetFileName)
```

```
/Y &#xD;&#xA;copy $(TargetPath) $(SolutionDir)MyExtendableApp\
$(TargetFileName) /Y" />
</Target>
```

Теперь после компиляции каждого проекта его сборка копируется также в целевой каталог приложения `MyExtendableApp`.

Создание решения и проектов с помощью Visual Studio

Вспомните, что по умолчанию среда Visual Studio назначает решению такое же имя, как у первого проекта, созданного в этом решении. Тем не менее, вы можете легко изменять имя решения.

Чтобы создать решение `ExtendableApp`, выберите в меню пункт File⇨New Project (Файл⇨Создать проект). В открывшемся диалоговом окне Add New Project (Добавление нового проекта) выберите элемент Class Library (Библиотека классов) и введите `CommonSnappableTypes` в поле Project name (Имя проекта). Прежде чем щелкать на кнопке Create (Создать), введите `ExtendableApp` в поле Solution name (Имя решения), как показано на рис. 17.3.

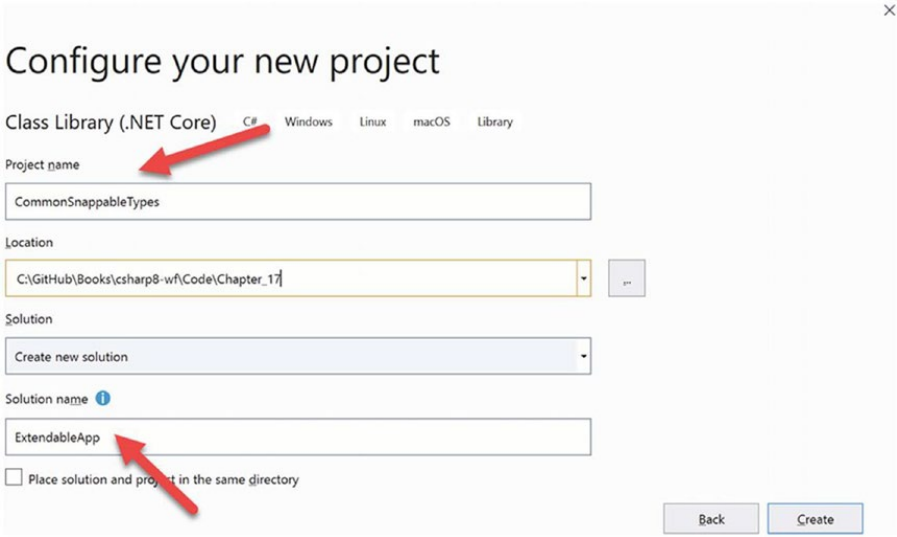

Рис. 17.3. Создание проекта `CommonSnappableTypes` и решения `ExtendableApp`

Чтобы добавить к решению еще один проект, щелкните правой кнопкой мыши на имени решения (`ExtendableApp`) в окне Solution Explorer и выберите в контекстном меню пункт Add⇨New Project (Добавить⇨Новый проект) или выберите в меню пункт File⇨Add⇨New Project (Файл⇨Добавить⇨Новый проект). При добавлении дополнительного проекта к существующему решению содержимое диалогового окна Add New Project слегка отличается; параметры решения теперь отсутствуют, так что вы увидите только информацию о проекте (имя и местоположение). Назначьте проекту библиотеки классов имя `CSharpSnapIn` и щелкните на кнопке Create.

Далее добавьте в проект CSharpSnapIn ссылку на проект CommonSnappableTypes. В среде Visual Studio щелкните правой кнопкой мыши на имени проекта CSharpSnapIn и выберите в контекстном меню пункт Add⇨Project Reference (Добавить⇨Ссылка на проект). В открывшемся диалоговом окне Reference Manager (Диспетчер ссылок) выберите элемент Projects⇨Solution (Проекты⇨Решение) в левой части (если он еще не выбран) и отметьте флажок рядом с CommonSnappableTypes.

Повторите процесс для нового проекта библиотеки классов Visual Basic (VBSnapIn), которая ссылается на проект CommonSnappableTypes.

Наконец, добавьте к решению новый проект консольного приложения .NET Core по имени MyExtendableApp. Добавьте в него ссылку на проект CommonSnappableTypes и установите проект консольного приложения в качестве стартового для решения. Для этого щелкните правой кнопкой мыши на имени проекта MyExtendableApp в окне Solution Explorer и выберите в контекстном меню пункт Set as Startup Project (Установить как стартовый проект).

На заметку! Если вы щелкнете правой кнопкой мыши на имени решения ExtendableApp, а не на имени одного из проектов, то в контекстном меню отобразится пункт Set StartUp Projects (Установить стартовые проекты). Помимо запуска по щелчку на кнопке Run (Запустить) только одного проекта можно настроить запуск множества проектов, что будет демонстрироваться позже в книге.

Установка зависимостей проектов при компиляции

Когда среде Visual Studio поступает команда запустить решение, стартовый проект и все проекты, на которые имеются ссылки, компилируются в случае обнаружения любых изменений; однако проекты, ссылки на которые отсутствуют, *не* компилируются. Положение дел можно изменить, устанавливая зависимости проектов. Для этого щелкните правой кнопкой мыши на имени решения в окне Solution Explorer, выберите в контекстном меню пункт Project Build Order (Порядок компиляции проектов), в открывшемся диалоговом окне перейдите на вкладку Dependencies (Зависимости) и в раскрывающемся списке Projects (Проекты) выберите MyExtendableApp.

Обратите внимание, что проект CommonSnappableTypes уже выбран и связанный с ним флажок отключен. Причина в том, что на него производится ссылка напрямую. Отметьте также флажки для проектов CSharpSnapIn и VBSnapIn (рис. 17.4).

Теперь при каждой компиляции проекта MyExtendableApp будут также компилироваться проекты CSharpSnapIn и VBSnapIn.

Добавление событий PostBuild

Откройте окно свойств проекта для CSharpSnapIn (щелкнув правой кнопкой мыши на имени проекта в окне Solution Explorer и выбрав в контекстном меню пункт Properties (Свойства)) и перейдите в нем на вкладку Build Events (События при компиляции). Щелкните на кнопке Edit Post-build (Редактировать события после компиляции) и затем щелкните на Macros>> (Макросы). Здесь вы можете видеть доступные для использования макросы, которые ссылаются на пути и/или имена файлов. Преимущество применения этих макросов в событиях, связанных с компиляцией, заключается в том, что они не зависят от машины и работают с относительными путями. Скажем, кто-то работает в каталоге по имени c-sharp-wf\code\chapter17. Вы можете работать в другом каталоге (вероятнее всего так и есть). За счет применения макросов инструмент MSBuild всегда будет использовать корректный путь относительно файлов *.csproj.

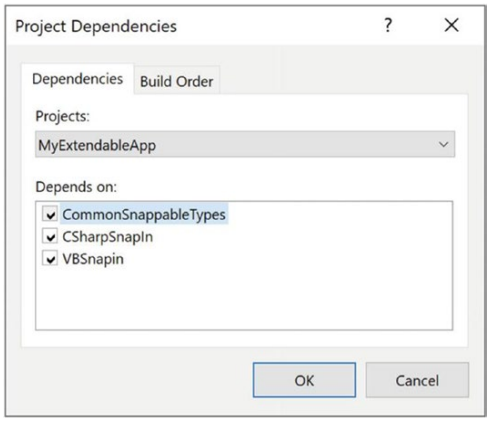

Рис. 17.4. Установка зависимостей проектов

Введите в области PostBuild (После компиляции) следующие две строки:

```
copy $(TargetPath) $(SolutionDir)MyExtendableApp\$(OutDir)
$(TargetFileName) /Y
copy $(TargetPath) $(SolutionDir)MyExtendableApp\$(TargetFileName) /Y
```

Сделайте то же самое для проекта `VBSnapIn`, но здесь вкладка в окне свойств называется Compile (Компиляция) и на ней понадобится щелкнуть на кнопке Build Events (События при компиляции).

Когда показанные выше команды событий после компиляции добавлены, все сборки при каждой компиляции будут копироваться в каталог проекта и выходной каталог приложения `MyExtendableApp`.

Построение сборки `CommonSnappableTypes.dll`

Удалите стандартный файл класса `Class1.cs` из проекта `CommonSnappableTypes`, добавьте новый файл интерфейса по имени `AppFunctionality.cs` и поместите в него следующий код:

```
namespace CommonSnappableTypes
{
  public interface IAppFunctionality
  {
    void DoIt();
  }
}
```

Добавьте файл класса по имени `CompanyInfoAttribute.cs` и приведите его содержимое к такому виду:

```
using System;
namespace CommonSnappableTypes
{
  [AttributeUsage(AttributeTargets.Class)]
```

```
  public sealed class CompanyInfoAttribute : System.Attribute
  {
    public string CompanyName { get; set; }
    public string CompanyUrl { get; set; }
  }
}
```

Тип `IAppFunctionality` обеспечивает полиморфный интерфейс для всех оснасток, которые могут потребляться расширяемым приложением. Учитывая, что рассматриваемый пример является полностью иллюстративным, в интерфейсе определен единственный метод под названием `DoIt()`.

Тип `CompanyInfoAttribute` — это специальный атрибут, который может применяться к любому классу, желающему подключиться к контейнеру. Как несложно заметить по определению класса, `[CompanyInfo]` позволяет разработчику оснастки указывать общие сведения о месте происхождения компонента.

Построение оснастки на C#

Удалите стандартный файл `Class1.cs` из проекта `CSharpSnapIn` и добавьте новый файл по имени `CSharpModule.cs`. Поместите в него следующий код:

```
using System;
using CommonSnappableTypes;
namespace CSharpSnapIn
{
  [CompanyInfo(CompanyName = "FooBar", CompanyUrl = "www.FooBar.com")]
  public class CSharpModule : IAppFunctionality
  {
    void IAppFunctionality.DoIt()
    {
      Console.WriteLine("You have just used the C# snap-in!");
    }
  }
}
```

Обратите внимание на явную реализацию интерфейса `IAppFunctionality` (см. главу 8). Поступать так необязательно; тем не менее, идея заключается в том, что единственной частью системы, которая нуждается в прямом взаимодействии с упомянутым интерфейсным типом, будет размещающее приложение. Благодаря явной реализации интерфейса `IAppFunctionality` метод `DoIt()` не доступен напрямую из типа `CSharpModule`.

Построение оснастки на Visual Basic

Теперь перейдите к проекту `VBSnapIn`. Удалите файл `Class1.vb` и добавьте новый файл по имени `VBSnapIn.vb`. Код Visual Basic столь же прост:

```
Imports CommonSnappableTypes
<CompanyInfo(CompanyName:="Chucky's Software",
             CompanyUrl:="www.ChuckySoft.com")>
Public Class VBSnapIn
  Implements IAppFunctionality

  Public Sub DoIt() Implements CommonSnappableTypes.IAppFunctionality.DoIt
    Console.WriteLine("You have just used the VB snap in!")
  End Sub
End Class
```

Как видите, применение атрибутов в синтаксисе Visual Basic требует указания угловых скобок (<>), а не квадратных ([]). Кроме того, для реализации интерфейсных типов заданным классом или структурой используется ключевое слово `Implements`.

Добавление кода для `ExtendableApp`

Последним обновляемым проектом является консольное приложение C# (`MyExtendableApp`). После добавления к решению консольного приложения `MyExtendableApp` и установки его как стартового проекта добавьте ссылку на проект `CommonSnappableTypes`, но *не* на `CSharpSnapIn.dll` или `VbSnapIn.dll`.

Модифицируйте операторы `using` в начале файла `Program.cs`, как показано ниже:

```
using System;
using System.Linq;
using System.Reflection;
using CommonSnappableTypes;
```

Метод `LoadExternalModule()` выполняет следующие действия:

- динамически загружает в память выбранную сборку;
- определяет, содержит ли сборка типы, реализующие интерфейс `IAppFunctionality`;
- создает экземпляр типа, используя позднее связывание.

Если обнаружен тип, реализующий `IAppFunctionality`, тогда вызывается метод `DoIt()` и найденный тип передается методу `DisplayCompanyData()` для вывода дополнительной информации о нем посредством рефлексии.

```
static void LoadExternalModule(string assemblyName)
{
  Assembly theSnapInAsm = null;
  try
  {
    // Динамически загрузить выбранную сборку.
    theSnapInAsm = Assembly.LoadFrom(assemblyName);
  }
  catch (Exception ex)
  {
    // Ошибка при загрузке оснастки.
    Console.WriteLine($"An error occurred loading the snapin: {ex.Message}");
    return;
  }
  // Получить все совместимые с IAppFunctionality классы в сборке.
  var theClassTypes = theSnapInAsm
    .GetTypes()
    .Where(t => t.IsClass && (t.GetInterface("IAppFunctionality") != null))
      .ToList();
  if (!theClassTypes.Any())
  {
    Console.WriteLine("Nothing implements IAppFunctionality!");
                    // Ни один класс не реализует IAppFunctionality!
  }
```

```
    // Создать объект и вызвать метод DoIt().
    foreach (Type t in theClassTypes)
    {
      // Использовать позднее связывание для создания экземпляра типа.
      IAppFunctionality itfApp
           = (IAppFunctionality) theSnapInAsm.CreateInstance(
        t.FullName, true);
      itfApp?.DoIt();
      // Отобразить информацию о компании.
      DisplayCompanyData(t);
    }
  }
```

Финальная задача связана с отображением метаданных, предоставляемых атрибутом `[CompanyInfo]`. Создайте метод `DisplayCompanyData()`, который принимает параметр `System.Type`:

```
  static void DisplayCompanyData(Type t)
  {
    // Получить данные [CompanyInfo].
    var compInfo = t
      .GetCustomAttributes(false)
      .Where(ci => (ci is CompanyInfoAttribute));

    // Отобразить данные.
    foreach (CompanyInfoAttribute c in compInfo)
    {
      Console.WriteLine($"More info about {c.CompanyName} can be found at
{c.CompanyUrl}");
    }
  }
```

Наконец, модифицируйте операторы верхнего уровня следующим образом:

```
  Console.WriteLine("***** Welcome to MyTypeViewer *****");
  string typeName = "";
  do
  {
    Console.WriteLine("\nEnter a snapin to load");
                      // Введите оснастку для загрузки
    Console.Write("or enter Q to quit: ");
                // или Q для завершения

    // Получить имя типа.
    typeName = Console.ReadLine();

    // Желает ли пользователь завершить работу?
    if (typeName.Equals("Q", StringComparison.OrdinalIgnoreCase))
    {
      break;
    }

    // Попытаться отобразить тип.
    try
    {
      LoadExternalModule(typeName);
    }
```

```
    catch (Exception ex)
    {
      // Найти оснастку не удалось.
      Console.WriteLine("Sorry, can't find snapin");
    }
  }
  while (true);
```

На этом создание примера расширяемого приложения завершено. Вы смогли увидеть, что представленные в главе приемы могут оказаться весьма полезными, и их применение не ограничивается только разработчиками инструментов.

Резюме

Рефлексия является интересным аспектом надежной объектно-ориентированной среды. В мире .NET Core службы рефлексии вращаются вокруг класса `System.Type` и пространства имен `System.Reflection`. Вы видели, что рефлексия — это процесс помещения типа под "увеличительное стекло" во время выполнения с целью выяснения его характеристик и возможностей.

Позднее связывание представляет собой процесс создания экземпляра типа и обращения к его членам без предварительного знания имен членов типа. Позднее связывание часто является прямым результатом динамической загрузки, которая позволяет программным образом загружать сборку .NET Core в память. На примере построения расширяемого приложения было продемонстрировано, что это мощный прием, используемый разработчиками инструментов, а также их потребителями.

Кроме того, в главе была исследована роль программирования на основе атрибутов. Снабжение типов атрибутами приводит к дополнению метаданных лежащей в основе сборки.

ГЛАВА 18

Динамические типы и среда DLR

В версии .NET 4.0 язык C# получил новое ключевое слово `dynamic`, которое позволяет внедрять в строго типизированный мир безопасности к типам, точек с запятой и фигурных скобок поведение, характерное для сценариев. Используя такую слабую типизацию, можно значительно упростить решение ряда сложных задач написания кода и получить возможность взаимодействия с несколькими динамическими языками (вроде IronRuby и IronPython), которые поддерживают .NET Core.

В настоящей главе вы узнаете о ключевом слове `dynamic` и о том, как слабо типизированные вызовы отображаются на корректные объекты в памяти с применением *исполняющей среды динамического языка* (Dynamic Language Runtime — DLR). После освоения служб, предлагаемых средой DLR, вы увидите примеры использования динамических типов для облегчения вызова методов с поздним связыванием (через службы рефлексии) и простого взаимодействия с унаследованными библиотеками COM.

На заметку! Не путайте ключевое слово `dynamic` языка C# с концепцией динамической сборки (объясняемой в главе 19). Хотя ключевое слово `dynamic` может применяться при построении динамической сборки, все же это две совершенно независимые концепции.

Роль ключевого слова `dynamic` языка C#

В главе 3 вы ознакомились с ключевым словом `var`, которое позволяет объявлять локальные переменные таким способом, что их действительные типы данных определяются на основе начального присваивания во время компиляции (вспомните, что результат называется *неявной типизацией*). После того как начальное присваивание выполнено, вы имеете строго типизированную переменную, и любая попытка присвоить ей несовместимое значение приведет к ошибке на этапе компиляции.

Чтобы приступить к исследованию ключевого слова `dynamic` языка C#, создайте новый проект консольного приложения по имени `DynamicKeyword`. Добавьте в класс `Program` следующий метод и удостоверьтесь, что финальный оператор кода на самом деле генерирует ошибку на этапе компиляции, если убрать символы комментария:

```
static void ImplicitlyTypedVariable()
{
  // Переменная a имеет тип List<int>.
  var a = new List<int> {90};
  // Этот оператор приведет к ошибке на этапе компиляции!
  // a = "Hello";
}
```

Использование неявной типизации лишь потому, что она возможна, некоторые считают плохим стилем (если известно, что необходима переменная типа `List<int>`, то так и следует ее объявлять). Однако, как было показано в главе 13, неявная типизация удобна в сочетании с LINQ, поскольку многие запросы LINQ возвращают перечисления анонимных классов (через проецирование), которые напрямую объявлять в коде C# невозможно. Тем не менее, даже в таких случаях неявно типизированная переменная фактически будет строго типизированной.

В качестве связанного замечания: в главе 6 упоминалось, что `System.Object` является изначальным родительским классом внутри инфраструктуры .NET Core и может представлять все, что угодно. Опять-таки, объявление переменной типа `object` в результате дает строго типизированный фрагмент данных, но то, на что указывает эта переменная в памяти, может отличаться в зависимости от присваиваемой ссылки. Чтобы получить доступ к членам объекта, на который указывает ссылка в памяти, понадобится выполнить явное приведение.

Предположим, что есть простой класс по имени `Person`, в котором определены два автоматических свойства (`FirstName` и `LastName`), инкапсулирующие данные `string`. Взгляните на следующий код:

```
static void UseObjectVariable()
{
  // Пусть имеется класс по имени Person.
  object o = new Person() { FirstName = "Mike", LastName = "Larson" };
  // Для получения доступа к свойствам Person
  // переменную o потребуется привести к Person.
  Console.WriteLine("Person's first name is {0}", ((Person)o).FirstName);
}
```

А теперь возвратимся к ключевому слову `dynamic`. С высокоуровневой точки значения ключевое слово `dynamic` можно трактовать как специализированную форму типа `System.Object` — в том смысле, что переменной динамического типа данных может быть присвоено любое значение. На первый взгляд это может привести к серьезной путанице, поскольку теперь получается, что доступны три способа определения данных, внутренний тип которых явно не указан в кодовой базе. Например, следующий метод:

```
static void PrintThreeStrings()
{
  var s1 = "Greetings";
  object s2 = "From";
  dynamic s3 = "Minneapolis";
  Console.WriteLine("s1 is of type: {0}", s1.GetType());
  Console.WriteLine("s2 is of type: {0}", s2.GetType());
  Console.WriteLine("s3 is of type: {0}", s3.GetType());
}
```

в случае вызова приведет к такому выводу:

```
s1 is of type: System.String
s2 is of type: System.String
s3 is of type: System.String
```

Динамическая переменная и переменная, объявленная неявно или через ссылку на `System.Object`, существенно отличаются тем, что динамическая переменная *не является строго типизированной*. Выражаясь по-другому, динамические данные

не типизированы статически. Для компилятора C# ситуация выглядит так, что элементу данных, объявленному с ключевым словом `dynamic`, можно присваивать вообще любое начальное значение, и на протяжении периода его существования взамен начального значения может быть присвоено любое новое (возможно, не связанное) значение. Рассмотрим показанный ниже метод и результирующий вывод:

```
static void ChangeDynamicDataType()
{
  // Объявить одиночный динамический элемент данных по имени t.
  dynamic t = "Hello!";
  Console.WriteLine("t is of type: {0}", t.GetType());
  t = false;
  Console.WriteLine("t is of type: {0}", t.GetType());
  t = new List<int>();
  Console.WriteLine("t is of type: {0}", t.GetType());
}
```

Вот вывод:

```
t is of type: System.String
t is of type: System.Boolean
t is of type: System.Collections.Generic.List`1[System.Int32]
```

Имейте в виду, что приведенный выше код успешно скомпилировался и дал бы идентичный результат, если бы переменная `t` была объявлена с типом `System.Object`. Однако, как вскоре будет показано, ключевое слово `dynamic` предлагает много дополнительных возможностей.

Вызов членов на динамически объявленных данных

Учитывая то, что динамическая переменная способна принимать идентичность любого типа на лету (подобно переменной типа `System.Object`), у вас может возникнуть вопрос о способе обращения к членам такой переменной (свойствам, методам, индексаторам, событиям и т.п.). С точки зрения синтаксиса отличий нет. Нужно просто применить операцию точки к динамической переменной, указать открытый член и предоставить любые аргументы (если они требуются).

Но (и это очень важное "но") допустимость указываемых членов компилятор проверять не будет! Вспомните, что в отличие от переменной, определенной с типом `System.Object`, динамические данные не являются статически типизированными. Вплоть до времени выполнения не будет известно, поддерживают ли вызываемые динамические данные указанный член, переданы ли корректные параметры, правильно ли записано имя члена, и т.д. Таким образом, хотя это может показаться странным, следующий метод благополучно скомпилируется:

```
static void InvokeMembersOnDynamicData()
{
  dynamic textData1 = "Hello";
  Console.WriteLine(textData1.ToUpper());
  // Здесь можно было бы ожидать ошибки на этапе компиляции!
  // Однако все компилируется нормально.
  Console.WriteLine(textData1.toupper());
  Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
}
```

Обратите внимание, что во втором вызове `WriteLine()` предпринимается попытка обращения к методу по имени `toupper()` на динамическом элементе данных (при записи имени метода использовался неправильный регистр символов; оно должно выглядеть как `ToUpper()`). Как видите, переменная `textData1` имеет тип `string`, а потому известно, что у этого типа отсутствует метод с именем, записанным полностью в нижнем регистре. Более того, тип `string` определенно не имеет метода по имени `Foo()`, который принимает параметры `int`, `string` и `DataTime`!

Тем не менее, компилятор C# ни о каких ошибках не сообщает. Однако если вызвать метод `InvokeMembersOnDynamicData()`, то возникнет ошибка времени выполнения с примерно таким сообщением:

```
Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
  'string' does not contain a definition for 'toupper'
Необработанное исключение: Microsoft.CSharp.RuntimeBinder.
  RuntimeBinderException: string не содержит определения для toupper
```

Другое очевидное отличие между обращением к членам динамических и строго типизированных данных связано с тем, что когда к элементу динамических данных применяется операция точки, ожидаемое средство IntelliSense среды Visual Studio *не* активизируется. Взамен IDE-среда позволит вводить любое имя члена, какое только может прийти вам на ум.

Отсутствие возможности доступа к средству IntelliSense для динамических данных должно быть понятным. Тем не менее, как вы наверняка помните, это означает необходимость соблюдения предельной аккуратности при наборе кода C# для таких элементов данных. Любая опечатка или символ в неправильном регистре внутри имени члена приведет к ошибке времени выполнения, в частности к генерации исключения типа `RuntimeBinderException`.

Класс `RuntimeBinderException` представляет ошибку, которая будет сгенерирована при попытке обращения к несуществующему члену динамического типа данных (как в случае `toupper()` и `Foo()`). Та же самая ошибка будет инициирована, если для члена, который существует, указаны некорректные данные параметров.

Поскольку динамические данные настолько изменчивы, любые обращения к членам переменной, объявленной с ключевым словом `dynamic`, могут быть помещены внутрь подходящего блока `try/catch` для элегантной обработки ошибок:

```
static void InvokeMembersOnDynamicData()
{
  dynamic textData1 = "Hello";
  try
  {
    Console.WriteLine(textData1.ToUpper());
    Console.WriteLine(textData1.toupper());
    Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
  }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Если вызвать метод `InvokeMembersOnDynamicData()` снова, то можно заметить, что вызов `ToUpper()` (обратите внимание на заглавные буквы "T" и "U") работает корректно, но затем на консоль выводится сообщение об ошибке:

```
HELLO
'string' does not contain a definition for 'toupper'
string не содержит определение для toupper
```

Конечно, процесс помещения всех динамических обращений к методам в блоки try/catch довольно утомителен. Если вы тщательно следите за написанием кода и передачей параметров, тогда поступать так необязательно. Однако перехват исключений удобен, когда вы заранее не знаете, присутствует ли интересующий член в целевом типе.

Область использования ключевого слова dynamic

Вспомните, что неявно типизированные данные (объявленные с ключевым словом var) возможны только для локальных переменных в области действия члена. Ключевое слово var никогда не может использоваться с возвращаемым значением, параметром или членом класса/структуры. Тем не менее, это не касается ключевого слова dynamic. Взгляните на следующее определение класса:

```
namespace DynamicKeyword
{
  class VeryDynamicClass
  {
    // Динамическое поле.
    private static dynamic _myDynamicField;
    // Динамическое свойство.
    public dynamic DynamicProperty { get; set; }
    // Динамический тип возврата и динамический тип параметра.
    public dynamic DynamicMethod(dynamic dynamicParam)
    {
      // Динамическая локальная переменная.
      dynamic dynamicLocalVar = "Local variable";
      int myInt = 10;
      if (dynamicParam is int)
      {
        return dynamicLocalVar;
      }
      else
      {
        return myInt;
      }
    }
  }
}
```

Теперь обращаться к открытым членам можно было бы ожидаемым образом; однако при работе с динамическими методами и свойствами нет полной уверенности в том, каким будет тип данных! По правде говоря, определение VeryDynamicClass может оказаться не особенно полезным в реальном приложении, но оно иллюстрирует область, где допускается применять ключевое слово dynamic.

Ограничения ключевого слова dynamic

Невзирая на то, что с использованием ключевого слова dynamic можно определять разнообразные сущности, с ним связаны некоторые ограничения. Хотя они не настолько впечатляющие, следует помнить, что элементы динамических данных при

вызове метода не могут применять лямбда-выражения или анонимные методы C#. Например, показанный ниже код всегда будет давать в результате ошибки, даже если целевой метод на самом деле принимает параметр типа делегата, который в свою очередь принимает значение `string` и возвращает `void`:

```
dynamic a = GetDynamicObject();

// Ошибка! Методы на динамических данных не могут использовать
// лямбда-выражения!
a.Method(arg => Console.WriteLine(arg));
```

Чтобы обойти упомянутое ограничение, понадобится работать с лежащим в основе делегатом напрямую, используя приемы из главы 12. Еще одно ограничение заключается в том, что динамический элемент данных не может воспринимать расширяющие методы (см. главу 11). К сожалению, сказанное касается также всех расширяющих методов из API-интерфейсов LINQ. Следовательно, переменная, объявленная с ключевым словом `dynamic`, имеет ограниченное применение в рамках LINQ to Objects и других технологий LINQ:

```
dynamic a = GetDynamicObject();

// Ошибка! Динамические данные не могут найти расширяющий метод Select()!
var data = from d in a select d;
```

Практическое использование ключевого слова `dynamic`

С учетом того, что динамические данные не являются строго типизированными, не проверяются на этапе компиляции, не имеют возможности запускать средство IntelliSense и не могут быть целью запроса LINQ, совершенно корректно предположить, что применение ключевого слова `dynamic` лишь по причине его существования представляет собой плохую практику программирования.

Тем не менее, в редких обстоятельствах ключевое слово `dynamic` может радикально сократить объем вводимого вручную кода. В частности, при построении приложения .NET Core, в котором интенсивно используется позднее связывание (через рефлексию), ключевое слово `dynamic` может сэкономить время на наборе кода. Аналогично при разработке приложения .NET Core, которое должно взаимодействовать с унаследованными библиотеками COM (вроде тех, что входят в состав продуктов Microsoft Office), за счет использования ключевого слова `dynamic` можно значительно упростить кодовую базу. В качестве финального примера можно привести веб-приложения, построенные с применением ASP.NET Core: они часто используют тип `ViewBag`, к которому также допускается производить доступ в упрощенной манере с помощью ключевого слова `dynamic`.

На заметку! Взаимодействие с COM является строго парадигмой Windows и исключает межплатформенные возможности из вашего приложения.

Как с любым "сокращением", прежде чем его использовать, необходимо взвесить все "за" и "против". Применение ключевого слова `dynamic` — компромисс между краткостью кода и безопасностью к типам. В то время как C# в своей основе является строго типизированным языком, динамическое поведение можно задействовать (или нет) от вызова к вызову. Всегда помните, что использовать ключевое слово `dynamic` необязательно. Тот же самый конечный результат можно получить, написав альтернативный код вручную (правда, обычно намного большего объема).

Роль исполняющей среды динамического языка

Теперь, когда вы лучше понимаете сущность "динамических данных", давайте посмотрим, как их обрабатывать. Начиная с версии .NET 4.0, общеязыковая исполняющая среда (Common Language Runtime — CLR) получила дополняющую среду времени выполнения, которая называется *исполняющей средой динамического языка* (Dynamic Language Runtime — DLR). Концепция "динамической исполняющей среды" определенно не нова. На самом деле ее много лет используют такие языки программирования, как JavaScript, LISP, Ruby и Python. Выражаясь кратко, динамическая исполняющая среда предоставляет динамическим языкам возможность обнаруживать типы полностью во время выполнения без каких-либо проверок на этапе компиляции.

На заметку! Хотя большая часть функциональных средств среды DLR была перенесена в .NET Core (начиная с версии 3.0), паритет в плане функциональности между DLR в .NET Core 5 и .NET 4.8 так и не был достигнут.

Если у вас есть опыт работы со строго типизированными языками (включая C# без динамических типов), тогда идея такой исполняющей среды может показаться неподходящей. В конце концов, вы обычно хотите выявлять ошибки на этапе компиляции, а не во время выполнения, когда только возможно. Тем не менее, динамические языки и исполняющие среды предлагают ряд интересных средств, включая перечисленные ниже.

- Чрезвычайно гибкая кодовая база. Можно проводить рефакторинг кода, не внося многочисленных изменений в типы данных.
- Простой способ взаимодействия с разнообразными типами объектов, которые построены на разных платформах и языках программирования.
- Способ добавления или удаления членов типа в памяти во время выполнения.

Одна из задач среды DLR заключается в том, чтобы позволить различным динамическим языкам работать с исполняющей средой .NET Core и предоставлять им возможность взаимодействия с другим кодом .NET Core. Двумя популярными динамическими языками, которые используют DLR, являются IronPython и IronRuby. Указанные языки находятся в "динамической вселенной", где типы определяются целиком во время выполнения. К тому же данные языки имеют доступ ко всему богатству библиотек базовых классов .NET Core. А еще лучше то, что благодаря наличию ключевого слова `dynamic` их кодовые базы могут взаимодействовать с языком C# (и наоборот).

На заметку! В настоящей главе вопросы применения среды DLR для интеграции с динамическими языками не обсуждаются.

Роль деревьев выражений

Для описания динамического вызова в нейтральных терминах среда DLR использует *деревья выражений*. Например, взгляните на следующий код C#:

```
dynamic d = GetSomeData();
d.SuperMethod(12);
```

В приведенном выше примере среда DLR автоматически построит дерево выражения, которое по существу гласит: "Вызвать метод по имени `SuperMethod()` на объ-

екте d, передав число 12 в качестве аргумента". Затем эта информация (формально называемая *полезной нагрузкой*) передается корректному связывателю времени выполнения, который может быть динамическим связывателем C# или (как вскоре будет объяснено) даже унаследованным объектом COM.

Далее запрос отображается на необходимую структуру вызовов для целевого объекта. Деревья выражений обладают одной замечательной характеристикой (помимо того, что их не приходится создавать вручную): они позволяют писать фиксированный оператор кода C# и не беспокоиться о том, какой будет действительная цель.

Динамический поиск в деревьях выражений во время выполнения

Как уже объяснялось, среда DLR будет передавать деревья выражений целевому объекту; тем не менее, на этот процесс отправки влияет несколько факторов. Если динамический тип данных указывает в памяти на объект COM, то дерево выражения отправляется реализации низкоуровневого интерфейса COM по имени `IDispatch`. Как вам может быть известно, упомянутый интерфейс представляет собой способ, которым COM внедряет собственный набор динамических служб. Однако объекты COM можно использовать в приложении .NET Core без применения DLR или ключевого слова `dynamic` языка C#. Тем не менее, такой подход (как вы увидите) сопряжен с написанием более сложного кода на C#.

Если динамические данные не указывают на объект COM, тогда дерево выражения может быть передано объекту, реализующему интерфейс `IDynamicObject`. Указанный интерфейс используется "за кулисами", чтобы позволить языку вроде IronRuby принимать дерево выражения DLR и отображать его на специфические средства языка Ruby.

Наконец, если динамические данные указывают на объект, который *не* является объектом COM и *не* реализует интерфейс `IDynamicObject`, то это нормальный повседневный объект .NET Core. В таком случае дерево выражения передается на обработку связывателю исполняющей среды C#. Процесс отображения дерева выражений на специфические средства платформы .NET Core вовлекает в дело службы рефлексии.

После того как дерево выражения обработано определенным связывателем, динамические данные преобразуются в реальный тип данных в памяти, после чего вызывается корректный метод со всеми необходимыми параметрами. Теперь давайте рассмотрим несколько практических применений DLR, начав с упрощения вызовов .NET Core с поздним связыванием.

Упрощение вызовов с поздним связыванием посредством динамических типов

Одним из случаев, когда имеет смысл использовать ключевое слово `dynamic`, может быть работа со службами рефлексии, а именно — вызов методов с поздним связыванием. В главе 17 приводилось несколько примеров, когда вызовы методов такого рода могут быть полезными — чаще всего при построении расширяемого приложения. Там вы узнали, как применять метод `Activator.CreateInstance()` для создания объекта типа, о котором ничего не известно на этапе компиляции (помимо его отображаемого имени). Затем с помощью типов из пространства имен `System.Reflection` можно обращаться к членам объекта через механизм позднего связывания. Вспомните показанный ниже пример из главы 17:

```
static void CreateUsingLateBinding(Assembly asm)
{
  try
  {
    // Получить метаданные для типа MiniVan.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");
    // Создать экземпляр MiniVan на лету.
    object obj = Activator.CreateInstance(miniVan);
    // Получить информацию о TurboBoost.
    MethodInfo mi = miniVan.GetMethod("TurboBoost");
    // Вызвать метод (null означает отсутствие параметров).
    mi.Invoke(obj, null);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

В то время как приведенный код функционирует ожидаемым образом, нельзя не отметить его некоторую громоздкость. Вы должны вручную работать с классом `MethodInfo`, вручную запрашивать метаданные и т.д. В следующей версии того же метода используется ключевое слово `dynamic` и среда DLR:

```
static void InvokeMethodWithDynamicKeyword(Assembly asm)
{
  try
  {
    // Получить метаданные для типа Minivan.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");
    // Создать экземпляр MiniVan на лету и вызвать метод.
    dynamic obj = Activator.CreateInstance(miniVan);
    obj.TurboBoost();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

За счет объявления переменной `obj` с ключевым словом `dynamic` вся рутинная работа, связанная с рефлексией, перекладывается на DLR.

Использование ключевого слова `dynamic` для передачи аргументов

Полезность среды DLR становится еще более очевидной, когда нужно выполнять вызовы методов с поздним связыванием, которые принимают параметры. В случае применения "многословных" обращений к рефлексии аргументы нуждаются в упаковке внутрь массива экземпляров `object`, который передается методу `Invoke()` класса `MethodInfo`.

Чтобы проиллюстрировать использование, создайте новый проект консольного приложения C# по имени `LateBindingWithDynamic`. Добавьте к решению проект библиотеки классов под названием `MathLibrary`. Переименуйте первоначальный файл `Class1.cs` в проекте `MathLibrary` на `SimplaMath.cs` и реализуйте класс, как показано ниже:

```
namespace MathLibrary
{
  public class SimpleMath
  {
    public int Add(int x, int y)
    {
      return x + y;
    }
  }
}
```

Модифицируйте содержимое файла `MathLibrary.csproj` следующим образом (чтобы скомпилированная сборка копировалась в целевой каталог `LateBindingWithDynamic`):

```
<Target Name="PostBuild" AfterTargets="PostBuildEvent">
  <Exec Command=
   "copy $(TargetPath) $(SolutionDir)LateBindingWithDynamic\$(OutDir)
$(TargetFileName) /Y &#xD;&#xA;copy $(TargetPath)
$(SolutionDir)LateBindingWithDynamic\
$(TargetFileName) /Y" />
</Target>
```

На заметку! Если вы не знакомы с событиями при компиляции, тогда ищите подробные сведения в главе 17.

Теперь возвратитесь к проекту `LateBindingWithDynamic` и импортируйте пространства имен `System.Reflection` и `Microsoft.CSharp.RuntimeBinder` в файл `Program.cs`. Добавьте в класс `Program` следующий метод, который вызывает метод `Add()` с применением типичных обращений к API-интерфейсу рефлексии:

```
static void AddWithReflection()
{
  Assembly asm = Assembly.LoadFrom("MathLibrary");
  try
  {
    // Получить метаданные для типа SimpleMath.
    Type math = asm.GetType("MathLibrary.SimpleMath");
    // Создать объект SimpleMath на лету.
    object obj = Activator.CreateInstance(math);
    // Получить информацию о методе Add().
    MethodInfo mi = math.GetMethod("Add");
    // Вызвать метод (с параметрами).
    object[] args = { 10, 70 };
    Console.WriteLine("Result is: {0}", mi.Invoke(obj, args));
                  // Вывод результата.
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Ниже показано, как можно упростить предыдущую логику, используя ключевое слово dynamic:

```
private static void AddWithDynamic()
{
  Assembly asm = Assembly.LoadFrom("MathLibrary");
  try
  {
    // Получить метаданные для типа SimpleMath.
    Type math = asm.GetType("MathLibrary.SimpleMath");
    // Создать объект SimpleMath на лету.
    dynamic obj = Activator.CreateInstance(math);
    // Обратите внимание, насколько легко теперь вызывать метод Add().
    Console.WriteLine("Result is: {0}", obj.Add(10, 70));
  }
  catch (RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

В результате вызова обоих методов получается идентичный вывод. Однако в случае применения ключевого слова dynamic сокращается объем кода. Благодаря динамически определяемым данным вам больше не придется вручную упаковывать аргументы внутрь массива экземпляров object, запрашивать метаданные сборки либо иметь дело с другими деталями подобного рода. При построении приложения, в котором интенсивно используется динамическая загрузка и позднее связывание, экономия на кодировании со временем становится еще более ощутимой.

Упрощение взаимодействия с COM посредством динамических данных (только Windows)

Давайте рассмотрим еще один полезный сценарий для ключевого слова dynamic в рамках проекта взаимодействия с COM. Если у вас нет опыта разработки для COM, то имейте в виду, что скомпилированная библиотека COM содержит метаданные подобно библиотеке .NET Core, но ее формат совершенно другой. По указанной причине, когда программа .NET Core нуждается во взаимодействии с объектом COM, первым делом потребуется сгенерировать так называемую *сборку взаимодействия* (описанную ниже). Задача довольно проста.

На заметку! Если вы не устанавливали индивидуальный компонент Visual Studio Tools for Office (Инструменты Visual Studio для Office) или рабочую нагрузку Office/SharePoint development (Разработка для Office/SharePoint), то для проработки примеров в текущем разделе вам придется это сделать. Можете запустить программу установки и выбрать недостающий компонент или воспользоваться средством быстрого запуска Visual Studio (<Ctrl+Q>). Введите Visual Studio Tools for Office в поле быстрого запуска и выберите вариант Install (Установить).

Для начала создайте новый проект консольного приложения по имени ExportDataToOfficeApp, откройте диалоговое окно Add COM Reference (Добавление

ссылки COM), перейдите на вкладку COM и отыщите желаемую библиотеку COM, которой в данном случае является Microsoft Excel 16.0 Object Library (рис. 18.1).

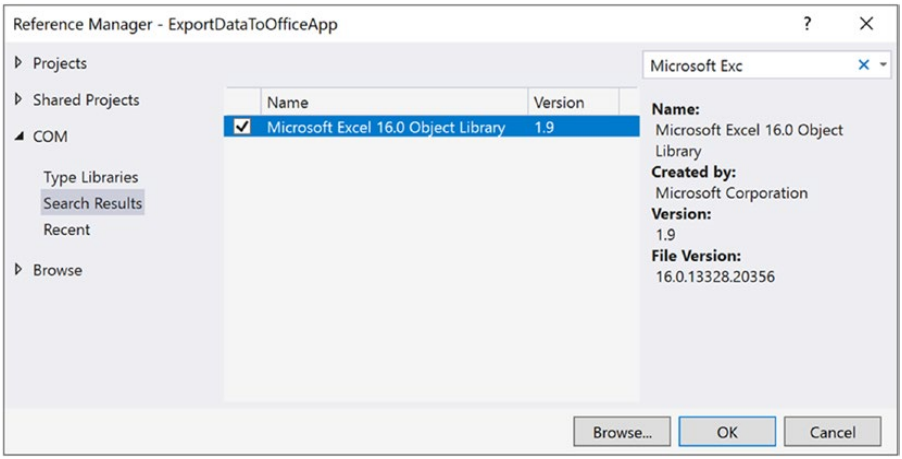

Рис. 18.1. На вкладке COM диалогового окна Add Reference отображаются все библиотеки COM, зарегистрированные на машине

После выбора COM-библиотеки IDE-среда отреагирует генерацией новой сборки, которая включает описания .NET Core метаданных COM. Формально она называется *сборкой взаимодействия* и не содержит какого-либо кода реализации кроме небольшой порции кода, который помогает транслировать события COM в события .NET Core. Тем не менее, сборки взаимодействия полезны тем, что защищают кодовую базу .NET Core от сложностей внутреннего механизма COM.

В коде C# можно напрямую работать со сборкой взаимодействия, которая отображает типы данных .NET Core на типы COM и наоборот. "За кулисами" данные маршализируются между приложениями .NET Core и COM с применением вызываемой оболочки времени выполнения (runtime callable wrapper — RCW), по существу являющейся динамически сгенерированным посредником. Такой посредник RCW будет маршализировать и трансформировать типы данных .NET Core в типы COM и отображать любые возвращаемые значения COM на их эквиваленты .NET Core.

Роль основных сборок взаимодействия

Многие библиотеки COM, созданные поставщиками библиотек COM (вроде библиотек Microsoft COM, обеспечивающих доступ к объектной модели продуктов Microsoft Office), предоставляют "официальную" сборку взаимодействия, которая называется *основной сборкой взаимодействия* (primary interop assembly — PIA). Сборки PIA — это оптимизированные сборки взаимодействия, которые приводят в порядок (и возможно расширяют) код, обычно генерируемый при добавлении ссылки на библиотеку COM с помощью диалогового окна Add Reference.

После добавления ссылки на библиотеку Microsoft Excel 16.0 Object Library просмотрите проект в окне Solution Explorer. Внутри узла Dependencies (Зависимости) вы увидите новый узел (COM) с элементом по имени `Interop.Microsoft.Office.Interop.Excel`. Это сгенерированный файл взаимодействия.

Встраивание метаданных взаимодействия

До выхода версии .NET 4.0, когда приложение C# задействовало библиотеку COM (через PIA или нет), на клиентской машине необходимо было обеспечить наличие копии сборки взаимодействия. Помимо увеличения размера установочного пакета приложения сценарий установки должен был также проверять, присутствуют ли сборки PIA, и если нет, тогда устанавливать их копии в глобальный кеш сборок (GAC).

На заметку! Глобальный кеш сборок был центральным хранилищем для сборок .NET Framework. В .NET Core он больше не используется.

Однако в .NET 4.0 и последующих версиях данные взаимодействия теперь можно встраивать прямо в скомпилированное приложение. В таком случае поставлять копию сборки взаимодействия вместе с приложением .NET Core больше не понадобится, т.к. все необходимые метаданные взаимодействия жестко закодированы в приложении .NET. В .NET Core встраивание сборки PIA является обязательным.

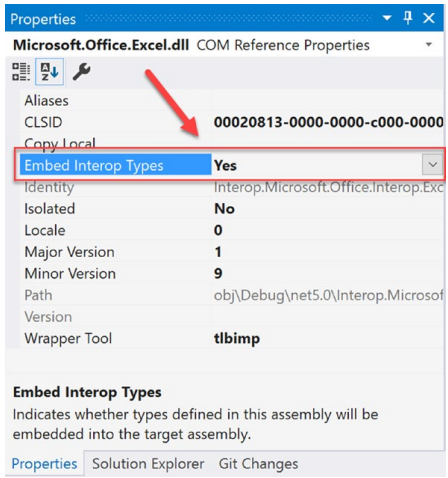

Рис. 18.2. Встраивание типов взаимодействия

Чтобы встроить сборку PIA в среде Visual Studio, разверните узел Dependencies внутри узла проекта, разверните узел COM, щелкните правой кнопкой мыши на элементе `Interop.Microsoft.Office.Interop.Excel` и выберите в контекстном меню пункт Properties (Свойства). В диалоговом окне Properties (Свойства) выберите в раскрывающемся списке Embed Interop Types (Встраивать типы взаимодействия) пункт Yes (Да), как показано на рис. 18.2.

Для изменения свойства посредством файла проекта добавьте узел `<EmbedInteropTypes>true</EmbedInteropTypes>`:

```
<ItemGroup>
  <COMReference Include="Microsoft.Office.Excel.dll">
    <Guid>00020813-0000-0000-c000-000000000046</Guid>
    <VersionMajor>1</VersionMajor>
    <VersionMinor>9</VersionMinor>
    <WrapperTool>tlbimp</WrapperTool>
    <Lcid>0</Lcid>
    <Isolated>false</Isolated>
    <EmbedInteropTypes>true</EmbedInteropTypes>
  </COMReference>
</ItemGroup>
```

Компилятор C# будет включать только те части библиотеки взаимодействия, которые вы используете. Таким образом, даже если реальная библиотека взаимодействия содержит описания .NET Core сотен объектов COM, в приложение попадет только под-

множество определений, которые действительно применяются в написанном коде C#. Помимо сокращения размера приложения, поставляемого клиенту, упрощается и процесс установки, т.к. не придется устанавливать сборки PIA, которые отсутствуют на целевой машине.

Общие сложности взаимодействия с COM

Многие библиотеки COM определяют методы, принимающие необязательные аргументы, которые вплоть до выхода .NET 3.5 в языке C# не поддерживались. Это требовало указания значения `Type.Missing` для каждого вхождения необязательного аргумента. К счастью, в .NET 3.5 и последующих версиях (включая .NET Core) значение `Type.Missing` вставляется на этапе компиляции, если не указано какое-то специфическое значение.

В качестве связанного замечания: многие методы COM поддерживают именованные аргументы, которые, как объяснялось в главе 4, позволяют передавать значения членам в любом порядке. Учитывая наличие поддержки той же самой возможности в языке C#, допускается просто "пропускать" необязательные аргументы, которые неважны, и устанавливать только те из них, которые нужны в текущий момент.

Еще одна распространенная сложность взаимодействия с COM была связана с тем фактом, что многие методы COM проектировались так, чтобы принимать и возвращать специфический тип данных по имени `Variant`. Во многом похоже на ключевое слово `dynamic` языка C#, типу данных `Variant` может быть присвоен на лету любой тип данных COM (строка, ссылка на интерфейс, числовое значение и т.д.). До появления ключевого слова `dynamic` передача и прием элементов данных типа `Variant` требовали немалых ухищрений, обычно связанных с многочисленными операциями приведения.

Когда свойство `EmbedInteropTypes` установлено в `true`, все COM-типы `Variant` автоматически отображаются на динамические данные. В итоге не только сокращается потребность в паразитных операциях приведения при работе с типами данных `Variant`, но также еще больше скрываются некоторые сложности, присущие COM, вроде работы с индексаторами COM.

Дополнительной сложностью при работе с взаимодействием с COM и .NET 5 является отсутствие поддержки на этапе компиляции и во время выполнения. Версия MSBuild в .NET 5 не способна распознавать библиотеки взаимодействия, поэтому проекты .NET Core, в которых задействовано взаимодействие с COM, не могут компилироваться с применением интерфейса командной строки .NET Core. Они должны компилироваться с использованием Visual Studio, и скомпилированный исполняющий файл можно будет запускать вполне ожидаемым способом.

Взаимодействие с COM с использованием динамических данных C#

Чтобы продемонстрировать, каким образом необязательные аргументы, именованные аргументы и ключевое слово `dynamic` совместно способствуют упрощению взаимодействия с COM, будет построено приложение, в котором применяется объектная модель Microsoft Office. Добавьте новый файл класса по имени `Car.cs`, содержащий такой код:

```
namespace ExportDataToOfficeApp
{
  public class Car
  {
    public string Make { get; set; }
    public string Color { get; set; }
    public string PetName { get; set; }
  }
}
```

Поместите в начало файла `Program.cs` следующие операторы `using`:

```
using System;
using System.Collections.Generic;
using System.Reflection;

// Создать псевдоним для объектной модели Excel.
using Excel = Microsoft.Office.Interop.Excel;
using ExportDataToOfficeApp;
```

Обратите внимание на псевдоним `Excel` для пространства имен `Microsoft.Office.Interop.Excel`. Хотя при взаимодействии с библиотеками COM псевдоним определять не обязательно, это обеспечивает наличие более короткого квалификатора для всех импортированных объектов COM. Он не только снижает объем набираемого кода, но также разрешает проблемы, когда объекты COM имеют имена, конфликтующие с именами типов .NET Core.

Далее создайте список записей `Car` в операторах верхнего уровня внутри файла `Program.cs`:

```
List<Car> carsInStock = new List<Car>
{
  new Car {Color="Green", Make="VW", PetName="Mary"},
  new Car {Color="Red", Make="Saab", PetName="Mel"},
  new Car {Color="Black", Make="Ford", PetName="Hank"},
  new Car {Color="Yellow", Make="BMW", PetName="Davie"}
};
```

Поскольку вы импортировали библиотеку COM с использованием Visual Studio, сборка PIA автоматически сконфигурирована так, что используемые метаданные будут встраиваться в приложение .NET Core. Таким образом, все типы данных `Variant` из COM реализуются как типы данных `dynamic`. Взгляните на показанную ниже реализацию метода `ExportToExcel()`:

```
void ExportToExcel(List<Car> carsInStock)
{
  // Загрузить Excel и затем создать новую пустую рабочую книгу.
  Excel.Application excelApp = new Excel.Application();
  excelApp.Workbooks.Add();

  // В этом примере используется единственный рабочий лист.
  Excel._Worksheet workSheet = (Excel._Worksheet)excelApp.ActiveSheet;

  // Установить заголовки столбцов в ячейках.
  workSheet.Cells[1, "A"] = "Make";
  workSheet.Cells[1, "B"] = "Color";
  workSheet.Cells[1, "C"] = "Pet Name";
```

```
    // Сопоставить все данные из List<Car> с ячейками электронной таблицы.
    int row = 1;
    foreach (Car c in carsInStock)
    {
      row++;
      workSheet.Cells[row, "A"] = c.Make;
      workSheet.Cells[row, "B"] = c.Color;
      workSheet.Cells[row, "C"] = c.PetName;
    }
    // Придать симпатичный вид табличным данным.
    workSheet.Range["A1"].AutoFormat(
      Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
    // Сохранить файл, завершить работу Excel и отобразить
    // сообщение пользователю.
    workSheet.SaveAs($@"{Environment.CurrentDirectory}\Inventory.xlsx");
    excelApp.Quit();
    Console.WriteLine("The Inventory.xslx file has been saved to your
app folder");
                    // Файл Inventory.xslx сохранен в папке приложения.
  }
```

Метод `ExportToExcel()` начинается с загрузки приложения Excel в память; однако на рабочем столе оно не отобразится. В данном приложении нас интересует только работа с внутренней объектной моделью Excel. Тем не менее, если необходимо отобразить пользовательский интерфейс Excel, тогда метод понадобится дополнить следующим кодом:

```
  void ExportToExcel(List<Car> carsInStock)
  {
    // Загрузить Excel и затем создать новую пустую рабочую книгу.
    Excel.Application excelApp = new Excel.Application();
    // Сделать пользовательский интерфейс Excel видимым на рабочем столе.
    excelApp.Visible = true;
    ...
  }
```

После создания пустого рабочего листа добавляются три столбца, именованные в соответствии со свойствами класса `Car`. Затем ячейки наполняются данными `List<Car>`, и файл сохраняется с жестко закодированным именем `Inventory.xlsx`.

Если вы запустите приложение, то сможете затем открыть файл `Inventory.xlsx`, который будет сохранен в подкаталоге `\bin\Debug\net5.0` вашего проекта.

Хотя не похоже, что в предыдущем коде использовались какие-либо динамические данные, имейте в виду, что среда DLR оказала значительную помощь. Без среды DLR код пришлось записывать примерно так:

```
  static void ExportToExcelManual(List<Car> carsInStock)
  {
    Excel.Application excelApp = new Excel.Application();
    // Потребуется пометить пропущенные параметры!
    excelApp.Workbooks.Add(Type.Missing);
```

```
    // Потребуется привести объект Object к _Worksheet!
    Excel._Worksheet workSheet =
      (Excel._Worksheet)excelApp.ActiveSheet;

    // Потребуется привести каждый объект Object к Range
    // и затем обратиться к низкоуровневому свойству Value2!
    ((Excel.Range)excelApp.Cells[1, "A"]).Value2 = "Make";
    ((Excel.Range)excelApp.Cells[1, "B"]).Value2 = "Color";
    ((Excel.Range)excelApp.Cells[1, "C"]).Value2 = "Pet Name";
    int row = 1;
    foreach (Car c in carsInStock)
    {
      row++;
      // Потребуется привести каждый объект Object к Range
      // и затем обратиться к низкоуровневому свойству Value2!
      ((Excel.Range)workSheet.Cells[row, "A"]).Value2 = c.Make;
      ((Excel.Range)workSheet.Cells[row, "B"]).Value2 = c.Color;
      ((Excel.Range)workSheet.Cells[row, "C"]).Value2 = c.PetName;
    }

    // Потребуется вызвать метод get_Range()
    // с указанием всех пропущенных аргументов!
    excelApp.get_Range("A1", Type.Missing).AutoFormat(
      Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2,
      Type.Missing, Type.Missing, Type.Missing,
      Type.Missing, Type.Missing, Type.Missing);

    // Потребуется указать все пропущенные необязательные аргументы!
    workSheet.SaveAs(
      $@"{Environment.CurrentDirectory}\InventoryManual.xlsx",
      Type.Missing, Type.Missing, Type.Missing,
      Type.Missing, Type.Missing, Type.Missing,
      Type.Missing, Type.Missing, Type.Missing);
    excelApp.Quit();
    Console.WriteLine("The InventoryManual.xslx file has been saved to
your app folder");
                    // Файл Inventory.xslx сохранен в папке приложения.
  }
```

На этом рассмотрение ключевого слова `dynamic` языка C# и среды DLR завершено. Вы увидели, насколько данные средства способны упростить сложные задачи программирования, и (что возможно даже важнее) уяснили сопутствующие компромиссы. Делая выбор в пользу динамических данных, вы теряете изрядную часть безопасности типов, и ваша кодовая база предрасположена к намного большему числу ошибок времени выполнения.

В то время как о среде DLR можно еще рассказать многое, основное внимание в главе было сосредоточено на темах, практичных и полезных при повседневном программировании. Если вы хотите изучить расширенные средства DLR, такие как интеграция с языками написания сценариев, тогда обратитесь в документацию по .NET Core (начните с поиска темы "Dynamic Language Runtime Overview" ("Обзор исполняющей среды динамического языка")).

Резюме

Ключевое слово `dynamic` позволяет определять данные, идентичность которых не известна вплоть до времени выполнения. При обработке исполняющей средой динамического языка (DLR) автоматически создаваемое "дерево выражения" передается подходящему связывателю динамического языка, а полезная нагрузка будет распакована и отправлена правильному члену объекта.

С применением динамических данных и среды DLR сложные задачи программирования на C# могут быть радикально упрощены, особенно действие по включению библиотек COM в состав приложений .NET Core. Было показано, что это предлагает несколько дальнейших упрощений взаимодействия с COM (которые не имеют отношения к динамическим данным), в том числе встраивание данных взаимодействия COM в разрабатываемые приложения, необязательные и именованные аргументы.

Хотя все рассмотренные средства определенно могут упростить код, всегда помните о том, что динамические данные существенно снижают безопасность к типам в коде C# и открывают возможности для возникновения ошибок времени выполнения. Тщательно взвешивайте все "за" и "против" использования динамических данных в своих проектах C# и надлежащим образом тестируйте их!

ГЛАВА 19

Язык CIL и роль динамических сборок

При построении полномасштабного приложения .NET Core вы почти наверняка будете использовать C# (или другой управляемый язык, такой как Visual Basic) из-за присущей ему продуктивности и простоты применения. Однако в начале книги было показано, что роль управляемого компилятора заключается в трансляции файлов кода `*.cs` в код CIL, метаданные типов и манифест сборки. Как выяснилось, CIL представляет собой полноценный язык программирования .NET Core, который имеет собственный синтаксис, семантику и компилятор (`ilasm.exe`).

В текущей главе будет предложен краткий экскурс по родному языку платформы .NET Core. Здесь вы узнаете о различиях между *директивой*, *атрибутом* и *кодом операции* CIL. Затем вы ознакомитесь с ролью возвратного проектирования сборок .NET Core и разнообразных инструментов программирования на CIL. Остаток главы посвящен основам определения пространств имен, типов и членов с использованием грамматики CIL. В завершение главы исследуется роль пространства имен `System.Reflection.Emit` и объясняется, как можно динамически конструировать сборки (с помощью инструкций CIL) во время выполнения.

Конечно, необходимость работать с низкоуровневым кодом CIL на повседневной основе будет возникать только у очень немногих программистов. Глава начинается с описания причин, по которым изучение синтаксиса и семантики такого языка .NET Core может оказаться полезным.

Причины для изучения грамматики языка CIL

Язык CIL является истинным родным языком платформы .NET Core. При построении сборки .NET с помощью выбранного управляемого языка (C#, VB, F# и т.д.) соответствующий компилятор транслирует исходный код в инструкции CIL. Подобно любому языку программирования CIL предлагает многочисленные лексемы, связанные со структурированием и реализацией. Поскольку CIL представляет собой просто еще один язык программирования .NET Core, не должен вызывать удивление тот факт, что сборки .NET Core можно создавать прямо на CIL и компилировать их посредством компилятора CIL (`ilasm.exe`).

На заметку! Как было указано в главе 1, ни `ildasm.exe`, ни `ilasm.exe` не поставляется вместе с исполняющей средой .NET 5. Получить эти инструменты можно двумя способами. Первый способ — скомпилировать .NET 5 Runtime из исходного кода, находящегося по ссылке `https://github.com/dotnet/runtime`. Второй и более простой способ —

загрузить желаемую версию из `www.nuget.org`. Инструмент `ildasm.exe` в хранилище NuGet доступен по ссылке `https://www.nuget.org/packages/Microsoft.NETCore.ILDAsm/`, а `ilasm.exe` — по ссылке `https://www.nuget.org/packages/Microsoft.NETCore.ILAsm/`. Убедитесь в том, что выбрали корректную версию (для данной книги необходима версия 5.0.0 или выше). Добавьте NuGet-пакеты `ILDasm` и `ILAsm` в свой проект с помощью следующих команд:

```
dotnet add package Microsoft.NETCore.ILDAsm --version 5.0.0
dotnet add package Microsoft.NETCore.ILAsm --version 5.0.0
```

Команды на самом деле не добавляют `ildasm.exe` или `ilasm.exe` в ваш проект, а помещают их в папку пакетов (в среде Windows):

```
%userprofile%\.nuget\packages\microsoft.netcore.ilasm
  \5.0.0\runtimes\native\
%userprofile%\.nuget\packages\microsoft.netcore.ildasm
  \5.0.0\runtimes\native\
```

Кроме того, оба инструмента версии 5.0.0 включены в папку `Chapter_19` внутри хранилища GitHub для настоящей книги.

Хотя и верно утверждение о том, что построением полного приложения .NET Core прямо на CIL занимаются лишь немногие программисты (если вообще такие есть), изучение этого языка все равно является чрезвычайно интересным занятием. Попросту говоря, чем лучше вы понимаете грамматику CIL, тем больше способны погрузиться в мир расширенной разработки приложений .NET Core. Обратившись к конкретным примерам, можно утверждать, что разработчики, разбирающиеся в CIL, обладают следующими навыками.

- Умеют дизассемблировать существующую сборку .NET Core, редактировать код CIL в ней и заново компилировать модифицированную кодовую базу в обновленный двоичный файл .NET Core. Скажем, некоторые сценарии могут требовать изменения кода CIL для взаимодействия с расширенными средствами COM.
- Умеют строить динамические сборки с применением пространства имен `System.Reflection.Emit`. Данный API-интерфейс позволяет генерировать в памяти сборку .NET Core, которая дополнительно может быть сохранена на диск. Это полезный прием для разработчиков инструментов, которым необходимо генерировать сборки на лету.
- Понимают аспекты CTS, которые не поддерживаются высокоуровневыми управляемыми языками, но существуют на уровне CIL. На самом деле CIL является единственным языком .NET Core, который позволяет получать доступ ко всем аспектам CTS. Например, за счет использования низкоуровневого кода CIL появляется возможность определения членов и полей глобального уровня (которые не разрешены в C#).

Ради полной ясности нужно еще раз подчеркнуть, что овладеть мастерством работы с языком C# и библиотеками базовых классов .NET Core можно и *без* изучения деталей кода CIL. Во многих отношениях знание CIL аналогично знанию языка ассемблера программистом на C (и C++). Те, кто разбирается в низкоуровневых деталях, способны создавать более совершенные решения поставленных задач и глубже понимают лежащую в основе среду программирования (и выполнения). Словом, если вы готовы принять вызов, тогда давайте приступим к исследованию внутренних деталей CIL.

На заметку! Имейте в виду, что эта глава не планировалась быть всеобъемлющим руководством по синтаксису и семантике CIL.

Директивы, атрибуты и коды операций CIL

Когда вы начинаете изучение низкоуровневых языков, таких как CIL, то гарантированно встретите новые (и часто пугающие) названия для знакомых концепций. Например, к этому моменту приведенный ниже набор элементов вы почти наверняка посчитаете ключевыми словами языка C# (и это правильно):

```
{new, public, this, base, get, set, explicit, unsafe, enum, operator, partial}
```

Тем не менее, внимательнее присмотревшись к элементам набора, вы сможете заметить, что хотя каждый из них действительно является ключевым словом C#, он имеет радикально отличающуюся семантику. Скажем, ключевое слово `enum` определяет производный от `System.Enum` тип, а ключевые слова `this` и `base` позволяют ссылаться на текущий объект и его родительский класс. Ключевое слово `unsafe` применяется для установления блока кода, который не может напрямую отслеживаться средой CLR, а ключевое слово `operator` дает возможность создать скрытый (специально именованный) метод, который будет вызываться, когда используется специфическая операция C# (такая как знак "плюс").

По разительному контрасту с высокоуровневым языком вроде C# в CIL не просто определен общий набор ключевых слов сам по себе. Напротив, набор лексем, распознаваемых компилятором CIL, подразделяется на следующие три обширные категории, основываясь на их семантике:

- директивы CIL;
- атрибуты CIL;
- коды операций CIL.

Лексемы CIL каждой категории выражаются с применением отдельного синтаксиса и комбинируются для построения допустимой сборки .NET Core.

Роль директив CIL

Прежде всего, существует набор хорошо известных лексем CIL, которые используются для описания общей структуры сборки .NET Core. Такие лексемы называются *директивами*. Директивы CIL позволяют информировать компилятор CIL о том, каким образом определять пространства имен, типы и члены, которые будут заполнять сборку.

Синтаксически директивы представляются с применением префикса в виде точки (`.`), например, `.namespace`, `.class`, `.publickeytoken`, `.override`, `.method`, `.assembly` и т.д. Таким образом, если в файле с расширением `*.il` (общепринятое расширение для файлов кода CIL) указана одна директива `.namespace` и три директивы `.class`, то компилятор CIL сгенерирует сборку, в которой определено единственное пространство имен .NET Core, содержащее три класса .NET Core.

Роль атрибутов CIL

Во многих случаях директивы CIL сами по себе недостаточно описательны для того, чтобы полностью выразить определение заданного типа .NET Core или члена типа. С учетом этого факта многие директивы CIL могут сопровождаться разнообразными *ат-*

рибутами CIL, которые уточняют способ обработки директивы. Например, директива `.class` может быть снабжена атрибутом `public` (для установления видимости типа), атрибутом `extends` (для явного указания базового класса типа) и атрибутом `implements` (для перечисления набора интерфейсов, поддерживаемых данным типом).

На заметку! Не путайте атрибут .NET Core (см. главу 17) и атрибут CIL, которые являются двумя совершенно разными понятиями.

Роль кодов операций CIL

После того как сборка, пространство имен и набор типов .NET Core определены в терминах языка CIL с использованием различных директив и связанных атрибутов, остается только предоставить логику реализации для типов. Это работа *кодов операций*. В традициях других низкоуровневых языков программирования многие коды операций CIL обычно имеют непонятный и совершенно нечитабельный вид. Например, для загрузки в память переменной типа `string` применяется код операции, который вместо дружественного имени наподобие `LoadString` имеет имя `ldstr`.

Справедливости ради следует отметить, что некоторые коды операций CIL довольно естественно отображаются на свои аналоги в C# (например, `box`, `unbox`, `throw` и `sizeof`). Вы увидите, что коды операций CIL всегда используются внутри области реализации члена и в отличие от директив никогда не записываются с префиксом-точкой.

Разница между кодами операций и их мнемоническими эквивалентами в CIL

Как только что объяснялось, для реализации членов отдельно взятого типа применяются коды операций вроде `ldstr`. Однако такие лексемы, как `ldstr`, являются *мнемоническими эквивалентами CIL* фактических *двоичных кодов операций CIL*. Чтобы выяснить разницу, напишите следующий метод C# в проекте консольного приложения .NET Core по имени `FirstSamples`:

```
int Add(int x, int y)
{
  return x + y;
}
```

В терминах CIL действие сложения двух чисел выражается посредством кода операции `0X58`. В том же духе вычитание двух чисел выражается с помощью кода операции `0X59`, а действие по размещению нового объекта в управляемой куче записывается с использованием кода операции `0X73`. С учетом описанной реальности "код CIL", обрабатываемый JIT-компилятором, представляет собой не более чем порцию двоичных данных.

К счастью, для каждого двоичного кода операции CIL предусмотрен соответствующий мнемонический эквивалент. Например, вместо кода `0X58` может применяться мнемонический эквивалент `add`, вместо `0X59` — `sub`, а вместо `0X73` — `newobj`. С учетом такой разницы между кодами операций и их мнемоническими эквивалентами декомпиляторы CIL, подобные `ildasm.exe`, транслируют двоичные коды операций сборки в соответствующие им мнемонические эквиваленты CIL. Вот как `ildasm.exe` представит в CIL предыдущий метод `Add()`, написанный на языке C# (в зависимости от версии .NET Core вывод может отличаться):

```
.method assembly hidebysig static int32 Add(int32 x,int32 y) cil managed
{
  // Code size 9 (0x9)
  .maxstack 2
  .locals init ([0] int32 int32 V_0)
  IL_0000:  /* 00   |                */ nop
  IL_0001:  /* 02   |                */ ldarg.0
  IL_0002:  /* 03   |                */ ldarg.1
  IL_0003:  /* 58   |                */ add
  IL_0004:  /* 0A   |                */ stloc.0
  IL_0005:  /* 2B   | 00             */ br.s       IL_0007
  IL_0007:  /* 06   |                */ ldloc.0
  IL_0008:  /* 2A   |                */ ret
} // end of method
```

Если вы не занимаетесь разработкой исключительно низкоуровневого программного обеспечения .NET Core (вроде специального управляемого компилятора), то иметь дело с числовыми двоичными кодами операций CIL никогда не придется. На практике когда программисты, использующие .NET Core, говорят о "кодах операций CIL", они имеют в виду набор дружественных строковых мнемонических эквивалентов (что и делается в настоящей книге), а не лежащие в основе числовые значения.

Заталкивание и выталкивание: основанная на стеке природа CIL

В языках .NET Core высокого уровня (таких как C#) предпринимается попытка насколько возможно скрыть из виду низкоуровневые детали CIL. Один из особенно хорошо скрываемых аспектов — тот факт, что CIL является языком программирования, основанным на использовании стека. Вспомните из исследования пространств имен коллекций (см. главу 10), что класс `Stack<T>` может применяться для помещения значения в стек, а также для извлечения самого верхнего значения из стека с целью последующего использования. Разумеется, программисты на языке CIL не работают с объектом типа `Stack<T>` для загрузки и выгрузки вычисляемых значений, но применяемый ими образ действий похож на заталкивание и выталкивание.

Формально сущность, используемая для хранения набора вычисляемых значений, называется *виртуальным стеком выполнения*. Вы увидите, что CIL предоставляет несколько кодов операций, которые служат для помещения значения в стек; такой процесс именуется *загрузкой*. Кроме того, в CIL определены дополнительные коды операций, которые перемещают самое верхнее значение из стека в память (скажем, в локальную переменную), применяя процесс под названием *сохранение*.

В мире CIL невозможно напрямую получать доступ к элементам данных, включая локально определенные переменные, входные аргументы методов и данные полей типа. Вместо этого элемент данных должен быть явно загружен в стек и затем извлекаться оттуда для использования в более позднее время (запомните упомянутое требование, поскольку оно содействует пониманию того, почему блок кода CIL может выглядеть несколько избыточным).

На заметку! Вспомните, что код CIL не выполняется напрямую, а компилируется по требованию. Во время компиляции кода CIL многие избыточные аспекты реализации оптимизируются. Более того, если для текущего проекта включена оптимизация кода (на вкладке Build (Сборка) окна свойств проекта в Visual Studio), то компилятор будет также удалять разнообразные избыточные детали CIL.

Чтобы понять, каким образом CIL задействует модель обработки на основе стека, создайте простой метод C# по имени `PrintMessage()`, который не принимает аргументов и возвращает `void`. Внутри его реализации будет просто выводиться значение локальной переменной в стандартный выходной поток:

```
void PrintMessage()
{
  string myMessage = "Hello.";
  Console.WriteLine(myMessage);
}
```

Если просмотреть код CIL, который получился в результате трансляции метода `PrintMessage()` компилятором C#, то первым делом обнаружится, что в нем определяется ячейка памяти для локальной переменной с помощью директивы `.locals`. Затем локальная строка загружается и сохраняется в этой локальной переменной с применением кодов операций `ldstr` (загрузить строку) и `stloc.0` (сохранить текущее значение в локальной переменной, находящейся в ячейке 0).

Далее с помощью кода операции `ldloc.0` (загрузить локальный аргумент по индексу 0) значение (по индексу 0) загружается в память для использования в вызове метода `System.Console.WriteLine()`, представленном кодом операции `call`. Наконец, посредством кода операции `ret` производится возвращение из функции. Ниже показан (прокомментированный) код CIL для метода `PrintMessage()` (ради краткости из листинга были удалены коды операций `nop`):

```
.method assembly hidebysig static void PrintMessage() cil managed
{
  .maxstack 1
  // Определить локальную переменную типа string (по индексу 0).
  .locals init ([0] string V_0)

  // Загрузить в стек строку со значением "Hello.".
  ldstr "Hello."

  // Сохранить строковое значение из стека в локальной переменной.
  stloc.0

  // Загрузить значение по индексу 0.
  ldloc.0

  // Вызвать метод с текущим значением.
  call void [System.Console]System.Console::WriteLine(string)
  ret
}
```

На заметку! Как видите, язык CIL поддерживает синтаксис комментариев в виде двойной косой черты (и вдобавок синтаксис `/*...*/`). Подобно компилятору C# компилятор CIL игнорирует комментарии в коде.

Теперь, когда вы знаете основы директив, атрибутов и кодов операций CIL, давайте приступим к практическому программированию на CIL, начав с рассмотрения темы возвратного проектирования.

Возвратное проектирование

В главе 1 было показано, как применять утилиту `ildasm.exe` для просмотра кода CIL, сгенерированного компилятором C#. Тем не менее, вы можете даже не подозревать, что эта утилита позволяет сбрасывать код CIL, содержащийся внутри загруженной в нее сборки, во внешний файл. Полученный подобным образом код CIL можно редактировать и компилировать заново с помощью компилятора CIL (`ilasm.exe`).

Выражаясь формально, такой прием называется *возвратным проектированием* и может быть полезен в избранных обстоятельствах, которые перечислены ниже.

- Вам необходимо модифицировать сборку, исходный код которой больше не доступен.
- Вы работаете с далеким от идеала компилятором языка .NET Core, который генерирует неэффективный (или явно некорректный) код CIL, поэтому нужно изменять кодовую базу.
- Вы конструируете библиотеку взаимодействия с COM и хотите учесть ряд атрибутов COM IDL, которые были утрачены во время процесса преобразования (такие как COM-атрибут `[helpstring]`).

Чтобы ознакомиться с процессом возвратного проектирования, создайте новый проект консольного приложения .NET Core на языке C# по имени `RoundTrip` посредством интерфейса командной строки .NET Core (CLI):

```
dotnet new console -lang c# -n RoundTrip -o .\RoundTrip -f net5.0
```

Модифицируйте операторы верхнего уровня, как показано ниже:

```
// Простое консольное приложение C#.
Console.WriteLine("Hello CIL code!");
Console.ReadLine();
```

Скомпилируйте программу с применением интерфейса CLI:

```
dotnet build
```

На заметку! Вспомните из главы 1, что результатом компиляции всех сборок .NET Core (библиотек классов и консольных приложений) будут файлы с расширением `*.dll`, которые выполняются с применением интерфейса .NET Core CLI. Нововведением .NET Core 3+ (и последующих версий) является то, что файл `dotnet.exe` копируется в выходной каталог и переименовывается согласно имени сборки. Таким образом, хотя выглядит так, что ваш проект был скомпилирован в `RoundTrip.exe`, на самом деле он компилируется в `RoundTrip.dll`, а файл `dotnet.exe` копируется в `RoundTrip.exe` вместе с обязательными аргументами командной строки, необходимыми для запуска `Roundtrip.dll`.

Запустите `ildasm.exe` в отношении `RoundTrip.dll`, используя следующую команду (на уровне каталога решения):

```
ildasm /all /METADATA /out=.\RoundTrip\RoundTrip.il
  .\RoundTrip\bin\Debug\net5.0\RoundTrip.dll
```

На заметку! При сбрасывании содержимого сборки в файл утилита `ildasm.exe` также генерирует файл `*.res`. Такие ресурсные файлы можно игнорировать (и удалять), поскольку в текущей главе они не применяются. В них содержится низкоуровневая информация, касающаяся безопасности CLR (помимо прочих данных).

Теперь можете просмотреть файл RoundTrip.il в любом текстовом редакторе. Вот его содержимое (для удобства оно слегка переформатировано и снабжено комментариями):

```
// Ссылаемые сборки.
.assembly extern System.Runtime
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A)
  .ver 5:0:0:0
}
.assembly extern System.Console
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  .ver 5:0:0:0
}
// Наша сборка.
.assembly RoundTrip
{
  ...
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module RoundTrip.dll
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003
.corflags 0x00000001
// Определение класса Program.
.class private abstract auto ansi beforefieldinit '<Program>$'
  extends [System.Runtime]System.Object
{
  .custom instance void [System.Runtime]System.Runtime.CompilerServices.
CompilerGeneratedAttribute::.ctor()
    = ( 01 00 00 00 )
  .method private hidebysig static void '<Main>$'(string[] args) cil managed
  {
    // Помечает этот метод как точку входа исполняемой сборки.
    .entrypoint
    .maxstack  8
    IL_0000:  ldstr "Hello CIL code!"
    IL_0005:  call  void [System.Console]System.Console::WriteLine(string)
    IL_000a:  nop
    IL_000b:  call  string [System.Console]System.Console::ReadLine()
    IL_0010:  pop
    IL_0011:  ret
  } // end of method '<Program>$'::'<Main>$'
} // end of class '<Program>$'
```

Обратите внимание, что файл *.il начинается с объявления всех внешних сборок, на которые ссылается текущая скомпилированная сборка. Если бы в вашей библиотеке классов использовались дополнительные типы из других ссылаемых сборок (помимо System.Runtime и System.Console), тогда вы обнаружили бы дополнительные директивы .assembly extern.

Далее следует формальное определение сборки RoundTrip.dll, описанное с применением разнообразных директив CIL (.module, .imagebase и т.д.).

После документирования внешних ссылаемых сборок и определения текущей сборки находится определение типа Program, созданное из операторов верхнего уровня. Обратите внимание, что директива .class имеет различные атрибуты (многие из которых необязательны) вроде приведенного ниже атрибута extends, который указывает базовый класс для типа:

```
.class private abstract auto ansi beforefieldinit '<Program>$'
  extends [System.Runtime]System.Object
{ ... }
```

Основной объем кода CIL представляет реализацию стандартного конструктора класса и автоматически сгенерированного метода Main(), которые оба определены (частично) посредством директивы .method. После того, как эти члены были определены с использованием корректных директив и атрибутов, они реализуются с применением разнообразных кодов операций.

Важно понимать, что при взаимодействии с типами .NET Core (такими как System.Console) в CIL *всегда* необходимо использовать полностью заданное имя типа. Более того, полностью заданное имя типа *всегда* должно предваряться префиксом в форме дружественного имени сборки, где определен тип (в квадратных скобках). Взгляните на следующую реализацию метода Main() в CIL:

```
.method private hidebysig static void '<Main>$'(string[] args) cil managed
{
  // Помечает этот метод как точку входа исполняемой сборки.
  .entrypoint
  .maxstack  8
  IL_0000:  ldstr "Hello CIL code!"
  IL_0005:  call  void [System.Console]System.Console::WriteLine(string)
  IL_000a:  nop
  IL_000b:  call  string [System.Console]System.Console::ReadLine()
  IL_0010:  pop
  IL_0011:  ret
} // end of method '<Program>$'::'<Main>$'
```

Роль меток в коде CIL

Вы определенно заметили, что каждая строка в коде реализации предваряется лексемой в форме IL_XXX: (например, IL_0000:, IL_0001: и т.д.). Такие лексемы называются *метками кода* и могут именоваться в любой выбранной вами манере (при условии, что они не дублируются внутри области действия члена). При сбросе содержимого сборки в файл утилита ildasm.exe автоматически генерирует метки кода, которые следуют соглашению об именовании вида IL_XXXX:. Однако их можно заменить более описательными маркерами, например:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 8
  Nothing_1: nop
  Load_String: ldstr "Hello CIL code!"
  PrintToConsole: call void [System.Console]System.Console::WriteLine(string)
  Nothing_2: nop
  WaitFor_KeyPress: call string [System.Console]System.Console::ReadLine()
```

```
  RemoveValueFromStack: pop
  Leave_Function: ret
}
```

В сущности, большая часть меток кода совершенно не обязательна. Единственный случай, когда метки кода по-настоящему необходимы, связан с написанием кода CIL, в котором используются разнообразные конструкции ветвления или организации циклов, т.к. с помощью меток можно указывать, куда должен быть направлен поток логики. В текущем примере все автоматически сгенерированные метки кода можно удалить безо всяких последствий:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 8
  nop
  ldstr "Hello CIL code!"
  call void [System.Console]System.Console::WriteLine(string)
  nop
  call string [System.Console]System.Console::ReadLine()
  pop
  ret
}
```

Взаимодействие с CIL: модификация файла `*.il`

Теперь, когда вы имеете представление о том, из чего состоит базовый файл CIL, давайте завершим эксперимент с возвратным проектированием. Цель здесь довольно проста: изменить сообщение, которое выводится в окно консоли. Вы можете делать что-то большее, скажем, добавлять ссылки на сборки или создавать новые классы и методы, но мы ограничимся простым примером.

Чтобы внести изменение, вам понадобится модифицировать текущую реализацию операторов верхнего уровня, созданную в виде метода `<Main>$()`. Отыщите этот метод в файле `*.il` и измените сообщение на `"Hello from altered CIL code!"`.

Фактически код CIL был модифицирован для соответствия следующему определению на языке C#:

```
static void Main(string[] args)
{
  Console.WriteLine("Hello from altered CIL code!");
  Console.ReadLine();
}
```

Компиляция кода CIL

Предшествующие версии .NET позволяли компилировать файлы `*.il` с применением утилиты `ilasm.exe`. В .NET Core положение дел изменилось. Для компиляции файлов `*.il` вы должны использовать тип проекта `Microsoft.NET.Sdk.IL` и на момент написания главы он все еще не был частью стандартного комплекта SDK.

Начните с создания нового каталога на своей машине. Создайте в этом каталоге файл `global.json`, который применяется к текущему каталогу и всем его подкаталогам. Он используется для определения версии комплекта SDK, которая будет задействована при запуске команд .NET Core CLI. Модифицируйте содержимое файла, как показано ниже:

```
{
  "msbuild-sdks": {
    "Microsoft.NET.Sdk.IL": "5.0.0-preview.1.20120.5"
  }
}
```

На следующем шаге создается файл проекта. Создайте файл по имени `RoundTrip.ilproj` и приведите его содержимое к следующему виду:

```
<Project Sdk="Microsoft.NET.Sdk.IL">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <MicrosoftNetCoreIlasmPackageVersion>
      5.0.0-preview.1.20120.5
    </MicrosoftNetCoreIlasmPackageVersion>
    <ProduceReferenceAssembly>false</ProduceReferenceAssembly>
  </PropertyGroup>
</Project>
```

Наконец, скопируйте созданный файл `RoundTrip.il` в каталог проекта. Скомпилируйте сборку с применением .NET Core CLI:

```
dotnet build
```

Результирующие файлы будут находиться, как обычно, в подкаталоге `bin\debug\net5.0`. На этом этапе новое приложение можно запустить. Разумеется, в окне консоли отобразится обновленное сообщение. Хотя приведенный простой пример не является особенно впечатляющим, он иллюстрирует один из сценариев применения возвратного проектирования на CIL.

Директивы и атрибуты CIL

Теперь, когда вы знаете, как преобразовывать сборки .NET Core в файлы `*.il` и компилировать файлы `*.il` в сборки, можете переходить к более детальному исследованию синтаксиса и семантики языка CIL. В последующих разделах будет поэтапно рассматриваться процесс создания специального пространства имен, содержащего набор типов. Тем не менее, для простоты типы пока не будут иметь логики реализации своих членов. Разобравшись с созданием простых типов, можете переключить внимание на процесс определения "реальных" членов с использованием кодов операций CIL.

Указание ссылок на внешние сборки в CIL

Скопируйте файлы `global.json` и `NuGet.config` из предыдущего примера в новый каталог проекта. Создайте новый файл проекта по имени `CILTypes.ilproj`, содержимое которого показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk.IL">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <MicrosoftNetCoreIlasmPackageVersion>
      5.0.0-preview.1.20120.5
    </MicrosoftNetCoreIlasmPackageVersion>
    <ProduceReferenceAssembly>false</ProduceReferenceAssembly>
  </PropertyGroup>
</Project>
```

Затем создайте в текстовом редакторе новый файл по имени CILTypes.il. Первой задачей в проекте CIL является перечисление внешних сборок, которые будут задействованы текущей сборкой. В рассматриваемом примере применяются только типы, находящиеся внутри сборки System.Runtime.dll. В новом файле понадобится указать директиву .assembly с уточняющим атрибутом external. При добавлении ссылки на сборку со строгим именем, подобную System.Runtime.dll, также должны быть указаны директивы .publickeytoken и .ver:

```
.assembly extern System.Runtime
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  .ver 5:0:0:0
}
```

Определение текущей сборки в CIL

Следующее действие заключается в определении создаваемой сборки с использованием директивы .assembly. В простейшем случае сборка может быть определена за счет указания дружественного имени двоичного файла:

```
// Наша сборка.
.assembly CILTypes { }
```

Хотя такой код действительно определяет новую сборку .NET Core, обычно внутрь объявления будут помещаться дополнительные директивы. В рассматриваемом примере определение сборки необходимо снабдить номером версии 1.0.0.0 посредством директивы .ver (обратите внимание, что числа в номере версии отделяются друг от друга *двоеточиями*, а не точками, как принято в C#):

```
// Наша сборка.
.assembly CILTypes
{
  .ver 1:0:0:0
}
```

Из-за того, что сборка CILTypes является однофайловой, ее определение завершается с применением следующей директивы .module, которая обозначает официальное имя двоичного файла .NET Core, т.е. CILTypes.dll:

```
// Наша сборка.
.assembly CILTypes
{
  .ver 1:0:0:0
}
// Модуль нашей однофайловой сборки.
.module CILTypes.dll
```

Кроме .assembly и .module существуют директивы CIL, которые позволяют дополнительно уточнять общую структуру создаваемого двоичного файла .NET Core. В табл. 19.1 перечислены некоторые наиболее распространенные директивы уровня сборки.

Таблица 19.1. Дополнительные директивы, связанные со сборками

Директива	Описание
.mresources	Если сборка использует внутренние ресурсы (такие как растровые изображения или таблицы строк), то данная директива применяется для указания имени файла, в котором содержатся ресурсы, подлежащие встраиванию в сборку
.subsystem	Эта директива CIL служит для указания предпочитаемого пользовательского интерфейса, в рамках которого желательно запускать сборку. Например, значение 2 указывает, что сборка должна выполняться в приложении с графическим пользовательским интерфейсом, а значение 3 — в консольном приложении

Определение пространств имен в CIL

Определив внешний вид и поведение сборки (а также обязательные внешние ссылки), вы можете создать пространство имен .NET Core (MyNamespace), используя директиву .namespace:

```
// Наша сборка имеет единственное пространство имен.
.namespace MyNamespace {}
```

Подобно C# определения пространств имен CIL могут быть вложены в другие пространства имен. Хотя здесь нет нужды определять корневое пространство имен, ради интереса посмотрим, как создать корневое пространство имен MyCompany:

```
.namespace MyCompany
{
  .namespace MyNamespace {}
}
```

Как и C#, язык CIL позволяет определить вложенное пространство имен следующим образом:

```
// Определение вложенного пространства имен.
.namespace MyCompany.MyNamespace {}
```

Определение типов классов в CIL

Пустые пространства имен не особо интересны, поэтому давайте рассмотрим процесс определения типов классов в CIL. Для определения нового типа класса предназначена директива .class. Тем не менее, эта простая директива может быть декорирована многочисленными дополнительными атрибутами, уточняющими природу типа. В целях иллюстрации добавим в наше пространство имен открытый класс под названием MyBaseClass. Как и в C#, если базовый класс явно не указан, то тип автоматически становится производным от System.Object:

```
.namespace MyNamespace
{
  // Предполагается базовый класс System.Object.
  .class public MyBaseClass {}
}
```

При построении типа, производного не от класса `System.Object`, применяется атрибут `extends`. Для ссылки на тип, определенный внутри той же самой сборки, язык CIL требует использования полностью заданного имени (однако если базовый тип находится внутри той же самой сборки, то префикс в виде дружественного имени сборки можно не указывать). Следовательно, демонстрируемая ниже попытка расширения `MyBaseClass` в результате дает ошибку на этапе компиляции:

```
// Этот код не скомпилируется!
.namespace MyNamespace
{
  .class public MyBaseClass {}
  .class public MyDerivedClass
    extends MyBaseClass {}
}
```

Чтобы корректно определить родительский класс для `MyDerivedClass`, потребуется указать полностью заданное имя `MyBaseClass`:

```
// Уже лучше!
.namespace MyNamespace
{
  .class public MyBaseClass {}
  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass {}
}
```

В дополнение к атрибутам `public` и `extends` определение класса CIL может иметь множество добавочных квалификаторов, которые управляют видимостью типа, компоновкой полей и т.д. В табл. 19.2 описаны избранные атрибуты, которые могут применяться в сочетании с директивой `.class`.

Таблица 19.2. Избранные атрибуты, используемые вместе с директивой `.class`

Атрибут	Описание
`public`, `private`, `nested assembly`, `nested famandassem`, `nested family`, `nested famorassem`, `nested public`, `nested private`	Эти атрибуты применяются для указания видимости заданного типа. Нетрудно заметить, что язык CIL предлагает много других возможностей помимо доступных в C#. За дополнительными сведениями обращайтесь в документ ECMA 335
`abstract`, `sealed`	Эти два атрибута могут быть присоединены к директиве `.class` для определения соответственно абстрактного или запечатанного класса
`auto`, `sequential`, `explicit`	Эти атрибуты инструктируют среду CLR о том, как размещать данные полей в памяти. Для типов классов подходит стандартный флаг `auto`. Изменение стандартной установки может быть удобно в случае использования P/Invoke для обращения к неуправляемому коду C
`extends`, `implements`	Эти атрибуты позволяют определять базовый класс для типа (посредством `extends`) и реализовывать интерфейс в типе (с помощью `implements`)

Определение и реализация интерфейсов в CIL

Несколько странно, но типы интерфейсов в CIL определяются с применением директивы `.class`. Тем не менее, когда директива `.class` декорирована атрибутом `interface`, тип трактуется как интерфейсный тип CTS. После определения интерфейс можно привязывать к типу класса или структуры с использованием атрибута `implements`:

```
.namespace MyNamespace
{
  // Определение интерфейса.
  .class public interface IMyInterface {}

  // Простой базовый класс.
  .class public MyBaseClass {}
  // Теперь MyDerivedClass реализует IMyInterface
  // и расширяет MyBaseClass.
  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass
    implements MyNamespace.IMyInterface {}
}
```

На заметку! Конструкция `extends` должна предшествовать конструкции `implements`. Кроме того, в конструкции `implements` может содержаться список интерфейсов с разделителями-запятыми.

Вспомните из главы 8, что интерфейсы могут выступать в роли базовых для других типов интерфейсов, позволяя строить иерархии интерфейсов. Однако вопреки возможным ожиданиям применять атрибут `extends` для порождения интерфейса `A` от интерфейса `B` в CIL нельзя. Атрибут `extends` используется только для указания базового класса типа. Когда интерфейс необходимо расширить, снова будет применяться атрибут `implements`, например:

```
// Расширение интерфейсов в CIL.
.class public interface IMyInterface {}

.class public interface IMyOtherInterface
  implements MyNamespace.IMyInterface {}
```

Определение структур в CIL

Директива `.class` может использоваться для определения любой структуры CTS, если тип расширяет `System.ValueType`. Кроме того, такая директива `.class` должна уточняться атрибутом `sealed` (учитывая, что структуры никогда не могут выступать в роли базовых для других типов значений). Если попытаться поступить иначе, тогда компилятор `ilasm.exe` выдаст сообщение об ошибке.

```
// Определение структуры всегда является запечатанным.
.class public sealed MyStruct
  extends [System.Runtime]System.ValueType{}
```

Имейте в виду, что в CIL предусмотрен сокращенный синтаксис для определения типа структуры. В случае применения атрибута `value` новый тип автоматически становится производным от `[System.Runtime]System.ValueType`. Следовательно, тип `MyStruct` можно было бы определить и так:

```
// Сокращенный синтаксис объявления структуры.
.class public sealed value MyStruct{}
```

Определение перечислений в CIL

Перечисления .NET Core порождены от класса `System.Enum`, который является `System.ValueType` (и потому также должен быть запечатанным). Чтобы определить перечисление в CIL, необходимо просто расширить `[System.Runtime]System.Enum`:

```
// Перечисление.
.class public sealed MyEnum
  extends [System.Runtime]System.Enum{}
```

Подобно структурам перечисления могут быть определены с помощью сокращенного синтаксиса, используя атрибут `enum`:

```
// Сокращенный синтаксис определения перечисления.
.class public sealed enum MyEnum{}
```

Вскоре вы увидите, как указывать пары "имя-значение" перечисления.

Определение обобщений в CIL

Обобщенные типы также имеют собственное представление в синтаксисе CIL. Вспомните из главы 10, что обобщенный тип или член может иметь один и более параметров типа. Например, в типе `List<T>` определен один параметр типа, а в `Dictionary<TKey, TValue>` — два. В CIL количество параметров типа указывается с применением символа обратной одиночной кавычки (`), за которым следует число, представляющее количество параметров типа. Как и в C#, действительные значения параметров типа заключаются в угловые скобки.

На заметку! На большинстве клавиатур символ ` находится на клавише, расположенной над клавишей <Tab> (и слева от клавиши <1>).

Например, предположим, что требуется создать переменную `List<T>`, где `T` — тип `System.Int32`. В C# пришлось бы написать такой код:

```
void SomeMethod()
{
  List<int> myInts = new List<int>();
}
```

В CIL необходимо поступить следующим образом (этот код может находиться внутри любого метода CIL):

```
// В C#: List<int> myInts = new List<int>();
newobj instance void class [System.Collections]
  System.Collections.Generic.List`1<int32>
  ::.ctor()
```

Обратите внимание, что обобщенный класс определен как `List`1<int32>`, поскольку `List<T>` имеет единственный параметр типа. А вот как определить тип `Dictionary<string, int>`:

```
// В C#: Dictionary<string, int> d = new Dictionary<string, int>();
newobj instance void class [System.Collections]
  System.Collections.Generic.Dictionary`2<string,int32>
  ::.ctor()
```

Рассмотрим еще один пример: пусть имеется обобщенный тип, использующий в качестве параметра типа другой обобщенный тип. Код CIL выглядит следующим образом:

```
// В C#: List<List<int>> myInts = new List<List<int>>();
newobj instance void class [System.Runtime]
  System.Collections.Generic.List`1<class
    [System.Collections]
    System.Collections.Generic.List`1<int32>>
    ::.ctor()
```

Компиляция файла CILTypes.il

Несмотря на то что к определенным ранее типам пока не были добавлены члены или код реализации, вы можете скомпилировать файл `*.il` в DLL-сборку .NET Core (так и нужно поступать ввиду отсутствия метода `Main()`). Откройте окно командной строки и введите показанную ниже команду:

```
dotnet build
```

Затем можете открыть скомпилированную сборку в `ildasm.exe`, чтобы удостовериться в создании каждого типа. Чтобы понять, каким образом заполнить тип содержимым, сначала необходимо ознакомиться с фундаментальными типами данных CIL.

Соответствия между типами данных в библиотеке базовых классов .NET Core, C# и CIL

В табл. 19.3 показано, как базовые классы .NET Core отображаются на соответствующие ключевые слова C#, а ключевые слова C# — на их представления в CIL. Кроме того, для каждого типа CIL приведено сокращенное константное обозначение. Как вы вскоре увидите, на такие константы часто ссылаются многие коды операций CIL.

Таблица 19.3. Отображение базовых классов .NET Core на ключевые слова C# и ключевых слов C# на CIL

Базовый класс .NET Core	Ключевое слово C#	Представление CIL	Константное обозначение CIL
System.SByte	sbyte	int8	I1
System.Byte	byte	unsigned int8	U1
System.Int16	short	int16	I2
System.UInt16	ushort	unsigned int16	U2
System.Int32	int	int32	I4
System.UInt32	uint	unsigned int32	U4
System.Int64	long	int64	I8
System.UInt64	ulong	unsigned int64	U8
System.Char	char	char	CHAR
System.Single	float	float32	R4
System.Double	double	float64	R8
System.Boolean	bool	bool	BOOLEAN
System.String	string	string	-
System.Object	object	object	-
System.Void	void	void	VOID

На заметку! Типы `System.IntPtr` и `System.UIntPtr` отображаются на собственные типы `int` и `unsigned int` в CIL (это полезно знать, т.к. они интенсивно применяются во многих сценариях взаимодействия с COM и P/Invoke).

Определение членов типов в CIL

Как вам уже известно, типы .NET Core могут поддерживать разнообразные члены. Перечисления содержат набор пар "имя-значение". Структуры и классы могут иметь конструкторы, поля, методы, свойства, статические члены и т.д. В предшествующих восемнадцати главах книги вы уже видели частичные определения в CIL упомянутых элементов, но давайте еще раз кратко повторим, каким образом различные члены отображаются на примитивы CIL.

Определение полей данных в CIL

Перечисления, структуры и классы могут поддерживать поля данных. Во всех случаях для их определения будет использоваться директива `.field`. Например, добавьте к перечислению `MyEnum` следующие три пары "имя-значение" (обратите внимание, что значения указаны в круглых скобках):

```
.class public sealed enum MyEnum
{
  .field public static literal valuetype
   MyNamespace.MyEnum A = int32(0)
  .field public static literal valuetype
   MyNamespace.MyEnum B = int32(1)
  .field public static literal valuetype
   MyNamespace.MyEnum C = int32(2)
}
```

Поля, находящиеся внутри области действия производного от `System.Enum` типа .NET Core, уточняются с применением атрибутов `static` и `literal`. Как не трудно догадаться, эти атрибуты указывают, что данные полей должны быть фиксированными значениями, доступными только из самого типа (например, `MyEnum.A`).

На заметку! Значения, присваиваемые полям в перечислении, также могут быть представлены в шестнадцатеричном формате с префиксом `0x`.

Конечно, когда нужно определить элемент поля данных внутри класса или структуры, вы не ограничены только открытыми статическими литеральными данными. Например, класс `MyBaseClass` можно было бы модифицировать для поддержки двух закрытых полей данных уровня экземпляра со стандартными значениями:

```
.class public MyBaseClass
{
  .field private string stringField = "hello!"
  .field private int32 intField = int32(42)
}
```

Как и в C#, поля данных класса будут автоматически инициализироваться подходящими стандартными значениями. Чтобы предоставить пользователю объекта возможность указывать собственные значения во время создания закрытых полей данных, потребуется создать специальные конструкторы.

Определение конструкторов типа в CIL

Спецификация CTS поддерживает создание конструкторов как уровня экземпляра, так и уровня класса (статических). В CIL конструкторы уровня экземпляра представляются с использованием лексемы `.ctor`, тогда как конструкторы уровня класса — посредством лексемы `.cctor` (class constructor — конструктор класса). Обе лексемы CIL должны сопровождаться атрибутами `rtspecialname` (return type special name — специальное имя возвращаемого типа) и `specialname`. Упомянутые атрибуты применяются для обозначения специфической лексемы CIL, которая может трактоваться уникальным образом в любом отдельно взятом языке .NET Core. Например, в языке C# конструкторы не определяют возвращаемый тип, но в CIL возвращаемым значением конструктора на самом деле является `void`:

```
.class public MyBaseClass
{
  .field private string stringField
  .field private int32 intField
  .method public hidebysig specialname rtspecialname
    instance void .ctor(string s, int32 i) cil managed
  {
    // Добавить код реализации...
  }
}
```

Обратите внимание, что директива `.ctor` снабжена атрибутом `instance` (поскольку конструктор не статический). Атрибуты `cil managed` указывают на то, что внутри данного метода содержится код CIL, а не неуправляемый код, который может использоваться при выполнении запросов P/Invoke.

Определение свойств в CIL

Свойства и методы также имеют специфические представления в CIL. В качестве примера модифицируйте класс `MyBaseClass` с целью поддержки открытого свойства по имени `TheString`, написав следующий код CIL (обратите внимание на применение атрибута `specialname`):

```
.class public MyBaseClass
{
  ...
  .method public hidebysig specialname
    instance string get_TheString() cil managed
  {
    // Добавить код реализации...
  }
  .method public hidebysig specialname
    instance void set_TheString(string 'value') cil managed
  {
    // Добавить код реализации...
  }
  .property instance string TheString()
  {
    .get instance string
      MyNamespace.MyBaseClass::get_TheString()
    .set instance void
      MyNamespace.MyBaseClass::set_TheString(string)
  }
}
```

В терминах CIL свойство отображается на пару методов, имеющих префиксы get_ и set_. В директиве .property используются связанные директивы .get и .set для отображения синтаксиса свойств на подходящие "специально именованные" методы.

На заметку! Обратите внимание, что входной параметр метода set в свойстве помещен в одинарные кавычки и представляет имя лексемы, которая должна применяться в правой части операции присваивания внутри области определения метода.

Определение параметров членов

Коротко говоря, параметры в CIL указываются (более или менее) идентично тому, как это делается в C#. Например, каждый параметр определяется путем указания его типа данных, за которым следует имя параметра. Более того, подобно C# язык CIL позволяет определять входные, выходные и передаваемые по ссылке параметры. Вдобавок в CIL допускается определять массив параметров (соответствует ключевому слову params в C#), а также необязательные параметры.

Чтобы проиллюстрировать процесс определения параметров в низкоуровневом коде CIL, предположим, что необходимо построить метод, который принимает параметр int32 (по значению), параметр int32 (по ссылке), параметр [System.Runtime.Extensions]System.Collection.ArrayList и один выходной параметр (типа int32). В C# метод выглядел бы примерно так:

```
public static void MyMethod(int inputInt,
  ref int refInt, ArrayList ar, out int outputInt)
{
  outputInt = 0;  // Просто чтобы удовлетворить компилятор C#...
}
```

После отображения метода MyMethod() на код CIL вы обнаружите, что ссылочные параметры C# помечаются символом амперсанда (&), который дополняет лежащий в основе тип данных (int32&).

Выходные параметры также снабжаются суффиксом &, но дополнительно уточняются лексемой [out] языка CIL. Вдобавок если параметр относится к ссылочному типу ([System.Runtime.Extensions]System.Collections.ArrayList), то перед типом данных указывается лексема class (не путайте ее с директивой .class):

```
.method public hidebysig static void MyMethod(int32 inputInt,
  int32& refInt,
  class [System.Runtime.Extensions]System.Collections.ArrayList ar,
  [out] int32& outputInt) cil managed
{
  ...
}
```

Исследование кодов операций CIL

Последний аспект кода CIL, который будет здесь рассматриваться, связан с ролью разнообразных кодов операций. Вспомните, что код операции — это просто лексема CIL, используемая при построении логики реализации для заданного члена.

Все коды операций CIL (которых довольно много) могут быть разделены на три обширные категории:

- коды операций, которые управляют потоком выполнения программы;
- коды операций, которые вычисляют выражения;
- коды операций, которые получают доступ к значениям в памяти (через параметры, локальные переменные и т.д.).

В табл. 19.4 описаны наиболее полезные коды операций, имеющие прямое отношение к логике реализации членов; они сгруппированы по функциональности.

Таблица 19.4. Различные коды операций CIL, связанные с реализацией членов

Коды операций	Описание
`add`, `sub`, `mul`, `div`, `rem`	Позволяют выполнять сложение, вычитание, умножение и деление двух значений (`rem` возвращает остаток от деления)
`and`, `or`, `not`, `xor`	Позволяют выполнять побитовые операции над двумя значениями
`ceq`, `cgt`, `clt`	Позволяют сравнивать два значения в стеке разными способами, например: `ceq` — сравнение на равенство `cgt` — сравнение "больше чем" `clt` — сравнение "меньше чем"
`box`, `unbox`	Применяются для преобразования между ссылочными типами и типами значений
`ret`	Используется для выхода из метода и возврата значения вызывающему коду (при необходимости)
`beq`, `bgt`, `ble`, `blt`, `switch`	Применяются для управления логикой ветвления внутри метода (вдобавок ко многим другим связанным кодам операций), например: `beq` — переход к метке в коде, если равно `bgt` — переход к метке в коде, если больше чем `ble` — переход к метке в коде, если меньше или равно `blt` — переход к метке в коде, если меньше чем Все коды операций, связанные с ветвлением, требуют указания в коде CIL метки для перехода в случае, если результат проверки оказывается истинным
`call`	Используется для вызова члена заданного типа
`newarr`, `newobj`	Позволяют размещать в памяти новый массив или новый объект (соответственно)

Коды операций из следующей обширной категории (подмножество которых описано в табл. 19.5) применяются для загрузки (заталкивания) аргументов в виртуальный стек выполнения. Обратите внимание, что все эти ориентированные на загрузку коды операций имеют префикс `ld` (load — загрузить).

Таблица 19.5. Основные коды операций CIL, связанные с загрузкой в стек

Код операции	Описание
`ldarg` (и многочисленные вариации)	Загружает в стек аргумент метода. Помимо общей формы `ldarg` (которая работает с индексом, идентифицирующим аргумент), существует множество других вариаций. Например, коды операций `ldarg`, которые имеют числовой суффикс (`ldarg.0`), жестко кодируют загружаемый аргумент. Кроме того, вариации `ldarg` позволяют жестко кодировать тип данных, используя константное обозначение CIL из табл. 19.3 (скажем, `ldarg_I4` для `int32`), а также тип данных и значение (например, `ldarg_I4_5` для загрузки `int32` со значением `5`)
`ldc` (и многочисленные вариации)	Загружает в стек константное значение
`ldfld` (и многочисленные вариации)	Загружает в стек значение поля уровня экземпляра
`ldloc` (и многочисленные вариации)	Загружает в стек значение локальной переменной
`ldobj`	Получает все значения, собранные размещенным в куче объектом, и помещает их в стек
`ldstr`	Загружает в стек строковое значение

В дополнение к набору кодов операций, связанных с загрузкой, CIL предоставляет многочисленные коды операций, которые явно извлекают из стека самое верхнее значение. Как было показано в нескольких начальных примерах, извлечение значения из стека обычно предусматривает его сохранение во временном локальном хранилище с целью дальнейшего использования (наподобие параметра для предстоящего вызова метода). Многие коды операций, извлекающие текущее значение из виртуального стека выполнения, снабжены префиксом `st` (store — сохранить). В табл. 19.6 описаны некоторые распространенные коды операций.

Таблица 19.6. Распространенные коды операций CIL, связанные с извлечением из стека

Код операции	Описание
`pop`	Удаляет значение, которое в текущий момент находится на верхушке стека вычислений, но не заботится о его сохранении
`starg`	Сохраняет значение из верхушки стека в аргументе метода с определенным индексом
`stloc` (и многочисленные вариации)	Извлекает текущее значение из верхушки стека вычислений и сохраняет его в списке локальных переменных по указанному индексу
`stobj`	Копирует значение указанного типа из стека вычислений по заданному адресу в памяти
`stsfld`	Заменяет значение статического поля значением из стека вычислений

Имейте в виду, что различные коды операций CIL будут неявно извлекать значения из стека во время выполнения своих задач. Например, при вычитании одного числа из другого с применением кода операции `sub` должно быть очевидным то, что перед самим вычислением операция `sub` должна извлечь из стека два следующих доступных значения. Результат вычисления снова помещается в стек.

Директива .maxstack

При написании кода реализации методов на низкоуровневом языке CIL необходимо помнить о специальной директиве под названием .maxstack. С ее помощью устанавливается максимальное количество переменных, которые могут находиться внутри стека в любой заданный момент времени на протяжении периода выполнения метода. Хорошая новость в том, что директива .maxstack имеет стандартное значение (8), которое должно подойти для подавляющего большинства создаваемых методов. Тем не менее, если вы хотите указывать все явно, то можете вручную подсчитать количество локальных переменных в стеке и определить это значение явно:

```
.method public hidebysig instance void
  Speak() cil managed
{
  // Внутри области действия этого метода в стеке находится
  // в точности одно значение (строковый литерал).
  .maxstack 1
  ldstr "Hello there..."
  call void [System.Runtime]System.Console::WriteLine(string)
  ret
}
```

Объявление локальных переменных в CIL

Теперь давайте посмотрим, как объявлять локальные переменные. Предположим, что необходимо построить в CIL метод по имени MyLocalVariables(), который не принимает аргументы и возвращает void, и определить в нем три локальные переменные с типами System.String, System.Int32 и System.Object. В C# такой метод выглядел бы следующим образом (вспомните, что локальные переменные не получают стандартные значения и потому перед использованием должны быть инициализированы):

```
public static void MyLocalVariables()
{
  string myStr = "CIL code is fun!";
  int myInt = 33;
  object myObj = new object();
}
```

А вот как реализовать метод MyLocalVariables() на языке CIL:

```
.method public hidebysig static void
  MyLocalVariables() cil managed
{
  .maxstack 8
  // Определить три локальные переменные.
  .locals init (string myStr, int32 myInt, object myObj)

  // Загрузить строку в виртуальный стек выполнения.
  ldstr "CIL code is fun!"

  // Извлечь текущее значение и сохранить его в локальной переменной [0].
  stloc.0

  // Загрузить константу типа i4 (сокращение для int32) со значением 33.
  ldc.i4.s 33
```

```
    // Извлечь текущее значение и сохранить его в локальной переменной [1].
    stloc.1
    // Создать новый объект и поместить его в стек.
    newobj instance void [System.Runtime]System.Object::.ctor()
    // Извлечь текущее значение и сохранить его в локальной переменной [2].
    stloc.2
    ret
}
```

Первым шагом при размещении локальных переменных с помощью CIL является применение директивы `.locals` в паре с атрибутом `init`. Каждая переменная идентифицируется своим типом данных и необязательным именем. После определения локальных переменных значения загружаются в стек (с использованием различных кодов операций загрузки) и сохраняются в этих локальных переменных (с помощью кодов операций сохранения).

Отображение параметров на локальные переменные в CIL

Вы уже видели, каким образом объявляются локальные переменные в CIL с применением директивы `.locals init`; однако осталось еще взглянуть на то, как входные параметры отображаются на локальные переменные. Рассмотрим показанный ниже статический метод C#:

```
public static int Add(int a, int b)
{
  return a + b;
}
```

Такой с виду невинный метод требует немалого объема кодирования на языке CIL. Во-первых, входные аргументы (`a` и `b`) должны быть помещены в виртуальный стек выполнения с использованием кода операции `ldarg` (load argument — загрузить аргумент). Во-вторых, с помощью кода операции `add` из стека будут извлечены следующие два значения и просуммированы с сохранением результата обратно в стек. В-третьих, сумма будет извлечена из стека и возвращена вызывающему коду посредством кода операции `ret`. Дизассемблировав этот метод C# с применением `ildasm.exe`, вы обнаружите множество дополнительных лексем, которые были внедрены в процессе компиляции, но основная часть кода CIL довольно проста:

```
.method public hidebysig static int32 Add(int32 a,
  int32 b) cil managed
{
  .maxstack 2
  ldarg.0      // Загрузить a в стек.
  ldarg.1      // Загрузить b в стек.
  add          // Сложить оба значения.
  ret
}
```

Скрытая ссылка `this`

Обратите внимание, что ссылка на два входных аргумента (`a` и `b`) в коде CIL производится с использованием их индексных позиций (`0` и `1`), т.к. индексация в виртуальном стеке выполнения начинается с нуля.

Во время исследования или написания кода CIL нужно помнить о том, что каждый нестатический метод, принимающий входные аргументы, автоматически получает неявный дополнительный параметр, который представляет собой ссылку на текущий объект (подобно ключевому слову `this` в C#). Скажем, если бы метод `Add()` был определен как *нестатический*:

```
// Больше не является статическим!
public int Add(int a, int b)
{
  return a + b;
}
```

то входные аргументы `a` и `b` загружались бы с применением кодов операций `ldarg.1` и `ldarg.2` (а не ожидаемых `ldarg.0` и `ldarg.1`). Причина в том, что ячейка 0 содержит неявную ссылку `this`. Взгляните на следующий псевдокод:

```
// Это ТОЛЬКО псевдокод!
.method public hidebysig static int32 AddTwoIntParams(
  MyClass_HiddenThisPointer this, int32 a, int32 b) cil managed
{
  ldarg.0  // Загрузить MyClass_HiddenThisPointer в стек.
  ldarg.1  // Загрузить a в стек.
  ldarg.2  // Загрузить b в стек.
  ...
}
```

Представление итерационных конструкций в CIL

Итерационные конструкции в языке программирования C# реализуются посредством ключевых слов `for`, `foreach`, `while` и `do`, каждое из которых имеет специальное представление в CIL. В качестве примера рассмотрим следующий классический цикл `for`:

```
public static void CountToTen()
{
  for(int i = 0; i < 10; i++)
  {
  }
}
```

Вспомните, что для управления прекращением потока выполнения, когда удовлетворено некоторое условие, используются коды операций `br` (`br`, `blt` и т.д.). В приведенном примере указано условие, согласно которому выполнение цикла `for` должно прекращаться, когда значение локальной переменной `i` становится больше или равно 10. С каждым проходом к значению `i` добавляется 1, после чего проверяемое условие оценивается заново.

Также вспомните, что в случае применения любого кода операции CIL, предназначенного для ветвления, должна быть определена специфичная метка кода (или две), обозначающая место, куда будет произведен переход при истинном результате оценки условия. С учетом всего сказанного рассмотрим показанный ниже (отредактированный) код CIL, который сгенерирован утилитой `ildasm.exe` (вместе с автоматически созданными метками):

```
.method public hidebysig static void CountToTen() cil managed
{
  .maxstack 2
  .locals init (int32 V_0, bool V_1)
  IL_0000: ldc.i4.0     // Загрузить это значение в стек.
  IL_0001: stloc.0      // Сохранить это значение по индексу 0.
  IL_0002: br.s IL_000b // Перейти на метку IL_0008.
  IL_0003: ldloc.0      // Загрузить значение переменной по индексу 0.
  IL_0004: ldc.i4.1     // Загрузить значение 1 в стек.
  IL_0005: add          // Добавить текущее значение в стеке по индексу 0
  IL_0006: stloc.0
  IL_0007: ldloc.0      // Загрузить значение по индексу 0.
  IL_0008: ldc.i4.s 10  // Загрузить значение 10 в стек.
  IL_0009: clt          // Меньше значения в стеке?
  IL_000a: stloc.1      // Сохранить результат по индексу 1.
  IL_000b: ldloc.1      // Загрузить значение переменной по индексу 1.
  IL_000c: brtrue.s IL_0002  // Если истинно, тогда перейти
                             //   на метку IL_0002.
  IL_000d: ret
}
```

Код CIL начинается с определения локальной переменной типа `int32` и ее загрузки в стек. Затем производятся переходы туда и обратно между метками `IL_0008` и `IL_0004`, во время каждого из которых значение `i` увеличивается на 1 и проверяется на предмет того, что оно все еще меньше 10. Как только условие будет нарушено, осуществляется выход из метода.

Заключительные слова о языке CIL

Ознакомившись с процессом создания исполняемого файла из файла `*.il`, вероятно у вас возникла мысль о том, что он требует чрезвычайно много работы и затем вопрос, в чем здесь выгода. В подавляющем большинстве случаев вы никогда не будете создавать исполняемый файл .NET Core из файла `*.il`. Тем не менее, способность понимать код CIL может принести пользу, когда вам нужно исследовать сборку, для которой отсутствует исходный код.

Существуют также коммерческие инструменты, которые восстанавливают исходный код сборки .NET Core. Если вам доводилось когда-либо пользоваться одним из инструментов подобного рода, то теперь вы знаете, каким образом они работают!

Динамические сборки

Естественно, процесс построения сложных приложений .NET Core на языке CIL будет довольно-таки неблагодарным трудом. С одной стороны, CIL является чрезвычайно выразительным языком программирования, который позволяет взаимодействовать со всеми программными конструкциями, разрешенными CTS. С другой стороны, написание низкоуровневого кода CIL утомительно, сопряжено с большими затратами времени и подвержено ошибкам. Хотя и правда, что знание — сила, вас может интересовать, насколько важно держать в уме все правила синтаксиса CIL. Ответ: зависит от ситуации. Разумеется, в большинстве случаев при программировании приложений .NET Core просматривать, редактировать или писать код CIL не потребуется. Однако знание основ языка CIL означает готовность перейти к исследованию мира динамических сборок (как противоположности статическим сборкам) и роли пространства имен `System.Reflection.Emit`.

Первым может возникнуть вопрос: чем отличаются статические сборки от динамических? По определению *статической сборкой* называется двоичная сборка .NET, которая загружается прямо из дискового хранилища, т.е. на момент запроса средой CLR она находится где-то на жестком диске в физическом файле (или в наборе файлов, если сборка многофайловая). Как и можно было предположить, при каждой компиляции исходного кода C# в результате получается статическая сборка.

Что касается *динамической сборки*, то она создается в памяти на лету с использованием типов из пространства имен `System.Reflection.Emit`, которое делает возможным построение сборки и ее модулей, определений типов и логики реализации на языке CIL *во время выполнения*. Затем сборку, расположенную в памяти, можно сохранить на диск, получив в результате новую статическую сборку. Ясно, что процесс создания динамических сборок с помощью пространства имен `System.Reflection.Emit` требует понимания природы кодов операций CIL.

Несмотря на то что создание динамических сборок является сложной (и редкой) задачей программирования, оно может быть удобным в разнообразных обстоятельствах. Ниже перечислены примеры.

- Вы строите инструмент программирования .NET Core, который должен быть способным генерировать сборки по требованию на основе пользовательского ввода.
- Вы создаете приложение, которое нуждается в генерации посредников для удаленных типов на лету, основываясь на полученных метаданных.
- Вам необходима возможность загрузки статической сборки и динамической вставки в двоичный образ новых типов.

Давайте посмотрим, какие типы доступны в пространстве имен `System.Reflection.Emit`.

Исследование пространства имен `System.Reflection.Emit`

Создание динамической сборки требует некоторых знаний кодов операций CIL, но типы из пространства имен `System.Reflection.Emit` максимально возможно скрывают сложность языка CIL. Скажем, вместо указания необходимых директив и атрибутов CIL для определения типа класса можно просто применять класс `TypeBuilder`. Аналогично, если нужно определить новый конструктор уровня экземпляра, то не придется задавать лексему `specialname`, `rtspecialname` или `.ctor`; взамен можно использовать класс `ConstructorBuilder`. Основные члены пространства имен `System.Reflection.Emit` описаны в табл. 19.7.

В целом типы из пространства имен `System.Reflection.Emit` позволяют представлять низкоуровневые лексемы CIL программным образом во время построения динамической сборки. Вы увидите многие из них в рассматриваемом далее примере; тем не менее, тип `ILGenerator` заслуживает специального внимания.

Роль типа `System.Reflection.Emit.ILGenerator`

Роль типа `ILGenerator` заключается во вставке кодов операций CIL внутрь заданного члена типа. Однако создавать объекты `ILGenerator` напрямую невозможно, т.к. этот тип не имеет открытых конструкторов. Взамен объекты `ILGenerator` должны получаться путем вызова специфических методов типов, относящихся к построителям (вроде `MethodBuilder` и `ConstructorBuilder`).

Таблица 19.7. Избранные члены пространства имен `System.Reflection.Emit`

Члены	Описание
`AssemblyBuilder`	Применяется для создания сборки (`*.dll` или `*.exe`) во время выполнения. В сборках `*.exe` должен вызываться метод `ModuleBuilder.SetEntryPoint()` для установки метода, который является точкой входа в модуль. Если ни одной точки входа не указано, тогда будет генерироваться файл `*.dll`
`ModuleBuilder`	Используется для определения набора модулей внутри текущей сборки
`EnumBuilder`	Применяется для создания типа перечисления .NET Core
`TypeBuilder`	Используется для создания классов, интерфейсов, структур и делегатов внутри модуля во время выполнения

`MethodBuilder`
`LocalBuilder`
`PropertyBuilder`
`FieldBuilder`
`ConstructorBuilder`
`CustomAttributeBuilder`
`ParameterBuilder`
`EventBuilder`
	Применяются для создания членов типов (таких как методы, локальные переменные, свойства, конструкторы и атрибуты) во время выполнения

`ILGenerator`	Выпускает коды операций CIL для указанного члена типа
`OpCodes`	Предоставляет многочисленные поля, которые отображаются на коды операций CIL. Используется вместе с разнообразными членами типа `System.Reflection.Emit.ILGenerator`

Вот пример:

```
// Получить объект ILGenerator из объекта ConstructorBuilder
// по имени myCtorBuilder.
ConstructorBuilder myCtorBuilder = /* */;
ILGenerator myCILGen = myCtorBuilder.GetILGenerator();
```

Имея объект `ILGenerator`, с помощью его методов можно выпускать низкоуровневые коды операций CIL. Некоторые (но не все) методы `ILGenerator` кратко описаны в табл. 19.8.

Основным методом класса `ILGenerator` является `Emit()`, который работает в сочетании с типом `System.Reflection.Emit.OpCodes`. Как упоминалось ранее в главе, данный тип открывает доступ к множеству полей только для чтения, которые отображаются на низкоуровневые коды операций CIL. Полный набор этих членов документирован в онлайновой справочной системе, и далее в главе вы неоднократно встретите примеры их использования.

Выпуск динамической сборки

Чтобы проиллюстрировать процесс определения сборки .NET Core во время выполнения, давайте рассмотрим процесс создания однофайловой динамической сборки по имени `MyAssembly.dll`.

Таблица 19.8. Избранные методы класса `ILGenerator`

Метод	Описание
`BeginCatchBlock()`	Начинает блок `catch`
`BeginExceptionBlock()`	Начинает блок обработки исключения
`BeginFinallyBlock()`	Начинает блок `finally`
`BeginScope()`	Начинает лексическую область
`DeclareLocal()`	Объявляет локальную переменную
`DefineLabel()`	Объявляет новую метку
`Emit()`	Многократно перегружен, чтобы позволить выпускать коды операций CIL
`EmitCall()`	Помещает в поток CIL код операции `call` или `callvirt`
`EmitWriteLine()`	Выпускает вызов `Console.WriteLine()` со значениями разных типов
`EndExceptionBlock()`	Завершает блок обработки исключения
`EndScope()`	Завершает лексическую область
`ThrowException()`	Выпускает инструкцию для генерации исключения
`UsingNamespace()`	Указывает пространство имен, которое должно применяться при оценке локальных и наблюдаемых значений в текущей активной лексической области

Внутри модуля находится класс `HelloWorld`, который поддерживает стандартный конструктор и специальный конструктор, применяемый для присваивания значения закрытой переменной-члена (`theMessage`) типа `string`. Вдобавок в классе `HelloWorld` имеется открытый метод экземпляра под названием `SayHello()`, который выводит приветственное сообщение в стандартный поток ввода-вывода, и еще один метод экземпляра по имени `GetMsg()`, возвращающий внутреннюю закрытую строку. По существу мы собираемся программно сгенерировать следующий тип класса:

```
// Этот класс будет создаваться во время выполнения с использованием
// пространства имен System.Reflection.Emit.
public class HelloWorld
{
  private string theMessage;
  HelloWorld() {}
  HelloWorld(string s) {theMessage = s;}
  public string GetMsg() {return theMessage;}
  public void SayHello()
  {
    System.Console.WriteLine("Hello from the HelloWorld class!");
  }
}
```

Создайте новый проект консольного приложения по имени `MyAsmBuilder` и добавьте NuGet-пакет `System.Reflection.Emit`. Импортируйте в него пространст-

ва имен `System.Reflection` и `System.Reflection.Emit`. Определите в классе `Program` статический метод по имени `CreateMyAsm()`. Этот единственный метод будет отвечать за решение следующих задач:

- определение характеристик динамической сборки (имя, версия и т.п.);
- реализация типа `HelloClass`;
- возвращение вызывающему методу объекта `AssemblyBuilder`.

Ниже приведен полный код, а затем его анализ:

```
static AssemblyBuilder CreateMyAsm()
{
  // Установить общие характеристики сборки.
  AssemblyName assemblyName = new AssemblyName
  {
    Name = "MyAssembly",
    Version = new Version("1.0.0.0")
  };
  // Создать новую сборку.
  var builder = AssemblyBuilder.DefineDynamicAssembly(
    assemblyName,AssemblyBuilderAccess.Run);
  // Определить имя модуля.
  ModuleBuilder module =
    builder.DefineDynamicModule("MyAssembly");
  // Определить открытый класс по имени HelloWorld.
  TypeBuilder helloWorldClass =
    module.DefineType("MyAssembly.HelloWorld",
    TypeAttributes.Public);
  // Определить закрытую переменную-член типа String по имени theMessage.
  FieldBuilder msgField = helloWorldClass.DefineField(
    "theMessage",
    Type.GetType("System.String"),
    attributes: FieldAttributes.Private);
  // Создать специальный конструктор.
  Type[] constructorArgs = new Type[1];
  constructorArgs[0] = typeof(string);
  ConstructorBuilder constructor =
    helloWorldClass.DefineConstructor(
      MethodAttributes.Public,
      CallingConventions.Standard,
      constructorArgs);
  ILGenerator constructorIl = constructor.GetILGenerator();
  constructorIl.Emit(OpCodes.Ldarg_0);
  Type objectClass = typeof(object);
  ConstructorInfo superConstructor =
    objectClass.GetConstructor(new Type[0]);
  constructorIl.Emit(OpCodes.Call, superConstructor);
  constructorIl.Emit(OpCodes.Ldarg_0);
  constructorIl.Emit(OpCodes.Ldarg_1);
  constructorIl.Emit(OpCodes.Stfld, msgField);
  constructorIl.Emit(OpCodes.Ret);
```

```
  // Создать стандартный конструктор.
  helloWorldClass.DefineDefaultConstructor(
    MethodAttributes.Public);
  // Создать метод GetMsg().
  MethodBuilder getMsgMethod = helloWorldClass.DefineMethod(
    "GetMsg",
    MethodAttributes.Public,
    typeof(string),
    null);
  ILGenerator methodIl = getMsgMethod.GetILGenerator();
  methodIl.Emit(OpCodes.Ldarg_0);
  methodIl.Emit(OpCodes.Ldfld, msgField);
  methodIl.Emit(OpCodes.Ret);
  // Создать метод SayHello().
  MethodBuilder sayHiMethod = helloWorldClass.DefineMethod(
    "SayHello", MethodAttributes.Public, null, null);
  methodIl = sayHiMethod.GetILGenerator();
  methodIl.EmitWriteLine("Hello from the HelloWorld class!");
  methodIl.Emit(OpCodes.Ret);
  // Выпустить класс HelloWorld.
  helloWorldClass.CreateType();
  return builder;
}
```

Выпуск сборки и набора модулей

Тело метода начинается с установления минимального набора характеристик сборки с применением типов `AssemblyName` и `Version` (определенных в пространстве имен `System.Reflection`). Затем производится получение объекта типа `AssemblyBuilder` через статический метод `AssemblyBuilder.DefineDynamicAssembly()`.

При вызове метода `DefineDynamicAssembly()` должен быть указан режим доступа к определяемой сборке, наиболее распространенные значения которого представлены в табл. 19.9.

Таблица 19.9. Часто используемые значения перечисления `AssemblyBuilderAccess`

Значение	Описание
`RunAndCollect`	Указывает, что сборка будет немедленно выгружена, а занимаемая ею память восстановлена, как только она перестанет быть доступной
`Run`	Указывает, что динамическая сборка может выполняться в памяти, но не сохраняться на диске

Следующая задача связана с определением набора модулей (и имени) для новой сборки. Метод `DefineDynamicModule()` возвращает ссылку на действительный объект типа `ModuleBuilder`:

```
// Создать новую сборку.
var builder = AssemblyBuilder.DefineDynamicAssembly(
  assemblyName,AssemblyBuilderAccess.Run);
```

Роль типа ModuleBuilder

Тип ModuleBuilder играет ключевую роль во время разработки динамических сборок. Как и можно было ожидать, ModuleBuilder поддерживает несколько членов, которые позволяют определять набор типов, содержащихся внутри модуля (классы, интерфейсы, структуры и т.д.), а также набор встроенных ресурсов (таблицы строк, изображения и т.п.). В табл. 19.10 описаны два метода, относящиеся к созданию. (Обратите внимание, что каждый метод возвращает объект связанного типа, который представляет тип, подлежащий созданию.)

Таблица 19.11. Избранные методы типа ModuleBuilder

Метод	Описание
DefineEnum()	Выпускает определение перечисления .NET Core
DefineType()	Конструирует объект TypeBuilder, который позволяет определять типы значений, интерфейсы и типы классов (включая делегаты)

Основным членом класса ModuleBuilder является метод DefineType(). Кроме указания имени типа (в виде простой строки) с помощью перечисления System.Reflection.TypeAttributes можно описывать формат этого типа. В табл. 19.11 приведены избранные члены перечисления TypeAttributes.

Таблица 19.11. Избранные члены перечисления TypeAttributes

Член	Описание
Abstract	Указывает, что тип является абстрактным
Class	Указывает, что тип является классом
Interface	Указывает, что тип является интерфейсом
NestedAssembly	Указывает, что класс находится в области видимости сборки, поэтому доступен только методам внутри его сборки
NestedFamANDAssem	Указывает, что класс находится в области видимости сборки и семейства, поэтому доступен только методам, которые относятся к пересечению семейства и сборки
NestedFamily	Указывает, что класс находится в области видимости семейства, поэтому доступен только методам, которые содержатся внутри собственного типа и любых подтипов
NestedFamORAssem	Указывает, что класс находится в области видимости семейства или сборки, поэтому доступен только методам, которые относятся к объединению семейства и сборки
NestedPrivate	Указывает, что класс является вложенным и закрытым
NestedPublic	Указывает, класс является вложенным и открытым
NotPublic	Указывает, что класс не является открытым
Public	Указывает, что класс является открытым
Sealed	Указывает, что класс является запечатанным и не может быть расширен
Serializable	Указывает, что класс может быть сериализирован

Выпуск типа HelloClass и строковой переменной-члена

Теперь, когда вы лучше понимаете роль метода ModuleBuilder.CreateType(), давайте посмотрим, как можно выпустить открытый тип класса HelloWorld и закрытую строковую переменную:

```
// Определить открытый класс по имени HelloWorld.
TypeBuilder helloWorldClass =
  module.DefineType("MyAssembly.HelloWorld",
  TypeAttributes.Public);

// Определить закрытую переменную-член типа String по имени theMessage.
FieldBuilder msgField = helloWorldClass.DefineField(
  "theMessage",
  Type.GetType("System.String"),
  attributes: FieldAttributes.Private);
```

Обратите внимание, что метод TypeBuilder.DefineField() предоставляет доступ к объекту типа FieldBuilder. В классе TypeBuilder также определены дополнительные методы, которые обеспечивают доступ к другим типам "построителей". Например, метод DefineConstructor() возвращает объект типа ConstructorBuilder, метод DefineProperty() — объект типа PropertyBuilder и т.д.

Выпуск конструкторов

Как упоминалось ранее, для определения конструктора текущего типа можно применять метод TypeBuilder.DefineConstructor(). Однако когда дело доходит до реализации конструктора HelloClass, в тело конструктора необходимо вставить низкоуровневый код CIL, который будет отвечать за присваивание входного параметра внутренней закрытой строке. Чтобы получить объект типа ILGenerator, понадобится вызвать метод GetILGenerator() из соответствующего типа "построителя" (в данном случае ConstructorBuilder).

Помещение кода CIL в реализацию членов осуществляется с помощью метода Emit() класса ILGenerator. В самом методе Emit() часто используется тип класса OpCodes, который открывает доступ к набору кодов операций CIL через свойства только для чтения. Например, свойство OpCodes.Ret обозначает возвращение из вызова метода, OpCodes.Stfld создает присваивание значения переменной-члену, а OpCodes.Call применяется для вызова заданного метода (конструктора базового класса в данном случае). Итак, логика для реализации конструктора будет выглядеть следующим образом:

```
// Создать специальный конструктор, принимающий
// единственный аргумент типа string.
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
  helloWorldClass.DefineConstructor(
    MethodAttributes.Public,
    CallingConventions.Standard,
    constructorArgs);

// Выпустить необходимый код CIL для конструктора.
ILGenerator constructorIl = constructor.GetILGenerator();
constructorIl.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
```

```
ConstructorInfo superConstructor =
  objectClass.GetConstructor(new Type[0]);
constructorIl.Emit(OpCodes.Call, superConstructor);

// Загрузить в стек указатель this объекта.
constructorIl.Emit(OpCodes.Ldarg_0);
constructorIl.Emit(OpCodes.Ldarg_1);

// Загрузить входной аргумент в виртуальный стек и сохранить его в msgField
constructorIl.Emit(OpCodes.Stfld, msgField);
constructorIl.Emit(OpCodes.Ret);
```

Как вам теперь уже известно, в результате определения специального конструктора для типа стандартный конструктор молча удаляется. Чтобы снова определить конструктор без аргументов, нужно просто вызвать метод `DefineDefaultConstructor()` типа `TypeBuilder`:

```
// Создать стандартный конструктор.
helloWorldClass.DefineDefaultConstructor(
  MethodAttributes.Public);
```

Выпуск метода `SayHello()`

В заключение давайте исследуем процесс выпуска метода `SayHello()`. Первая задача связана с получением объекта типа `MethodBuilder` из переменной `helloWorldClass`. После этого можно определить сам метод и получить внутренний объект типа `ILGenerator` для вставки необходимых инструкций CIL:

```
// Создать метод SayHello.
MethodBuilder sayHiMethod = helloWorldClass.DefineMethod(
  "SayHello", MethodAttributes.Public, null, null);
methodIl = sayHiMethod.GetILGenerator();

// Вывести строку на консоль.
methodIl.EmitWriteLine("Hello from the HelloWorld class!");
methodIl.Emit(OpCodes.Ret);
```

Здесь был определен открытый метод (т.к. указано значение `MethodAttributes.Public`), который не имеет параметров и ничего не возвращает (на что указывают значения `null` в вызове `DefineMethod()`). Также обратите внимание на вызов `EmitWriteLine()`. Посредством данного вспомогательного метода класса `ILGenerator` можно записать строку в стандартный поток вывода, приложив минимальные усилия.

Использование динамически сгенерированной сборки

Теперь, когда у вас есть логика для создания сборки, осталось лишь выполнить сгенерированный код. Логика в вызывающем коде обращается к методу `CreateMyAsm()`, получая ссылку на созданный объект `AssemblyBuilder`.

Далее вы поупражняетесь с поздним связыванием (см. главу 17) для создания экземпляра класса `HelloWorld` и взаимодействия с его членами. Модифицируйте операторы верхнего уровня, как показано ниже:

```
using System;
using System.Reflection;
using System.Reflection.Emit;
```

```
Console.WriteLine("***** The Amazing Dynamic Assembly Builder App *****");
// Создать объект AssemblyBuilder с использованием вспомогательной функции
AssemblyBuilder builder = CreateMyAsm();

// Получить тип HelloWorld.
Type hello = builder.GetType("MyAssembly.HelloWorld");

// Создать экземпляр HelloWorld и вызвать корректный конструктор.
Console.Write("-> Enter message to pass HelloWorld class: ");
string msg = Console.ReadLine();
object[] ctorArgs = new object[1];
ctorArgs[0] = msg;
object obj = Activator.CreateInstance(hello, ctorArgs);

// Вызвать метод SayHello() и отобразить возвращенную строку.
Console.WriteLine("-> Calling SayHello() via late binding.");
MethodInfo mi = hello.GetMethod("SayHello");
mi.Invoke(obj, null);

// Вызвать метод GetMsg().
mi = hello.GetMethod("GetMsg");
Console.WriteLine(mi.Invoke(obj, null));
```

Фактически только что была построена сборка .NET Core, которая способна создавать и запускать другие сборки .NET Core во время выполнения. На этом исследование языка CIL и роли динамических сборок завершено. Настоящая глава должна была помочь углубить знания системы типов .NET Core, синтаксиса и семантики языка CIL, а также способа обработки кода компилятором C# в процессе его компиляции.

Резюме

В главе был представлен обзор синтаксиса и семантики языка CIL. В отличие от управляемых языков более высокого уровня, таких как C#, в CIL не просто определяется набор ключевых слов, а предоставляются директивы (используемые для определения конструкции сборки и ее типов), атрибуты (дополнительно уточняющие данные директивы) и коды операций (применяемые для реализации членов типов).

Вы ознакомились с несколькими инструментами, связанными с программированием на CIL, и узнали, как изменять содержимое сборки .NET Core за счет добавления новых инструкций CIL, используя возвратное проектирование. Кроме того, вы изучили способы установления текущей (и ссылаемой) сборки, пространств имен, типов и членов. Был рассмотрен простой пример построения библиотеки кода и исполняемого файла .NET Core с применением CIL и соответствующих инструментов командной строки.

Наконец, вы получили начальное представление о процессе создания *динамической сборки*. Используя пространство имен `System.Reflection.Emit`, сборку .NET Core можно определять в памяти во время выполнения. Вы видели, что работа с этим пространством имен требует знания семантики кода CIL. Хотя построение динамических сборок не является распространенной задачей при разработке большинства приложений .NET Core, оно может быть полезно в случае создания инструментов поддержки и различных утилит для программирования.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Б

В

Г

Д

З

И

К

Л

P

C

Т

У

Ф

Ц

Ч

Я

У 10-му виданні книги описані новітні можливості мови C# 9 та .NET 5 разом із докладним "закулісним" обговоренням, покликаним розширити навички критичного мислення розробників, коли йдеться про їхнє ремесло. Книга охоплює ASP.NET Core, Entity Framework Core та багато іншого поряд з останніми оновленнями уніфікованої платформи. Усі приклади коду було переписано з урахуванням можливостей останнього випуску C#9.

Науково-популярне видання

Троелсен, Ендрю, **Джепікс**, Філіп

Мова програмування C# 9 і платформа .NET 5: основні принципи та практики програмування
Том 1, 10-е видання

(Рос. мовою)

Із загальних питань звертайтеся до видавництва "Діалектика" за адресою:
info@dialektika.com, http://www.dialektika.com

Підписано до друку 11.02.2022. Формат 60x90/16
Ум. друк. арк. 48,1. Обл.-вид. арк. 43,4

Видавець ТОВ "Комп'ютерне видавництво "Діалектика"
03164, м. Київ, вул. Генерала Наумова, буд. 23-Б.
Свідоцтво суб'єкта видавничої справи ДК № 6758 від 16.05.2019.

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

Язык программирования C# 9 и платформа .NET 5:
основные принципы и практики программирования

1 ТОМ

Эта классическая книга представляет собой всеобъемлющий источник сведений о языке программирования C# и о связанной с ним инфраструктуре. В 10-м издании книги вы найдете описание новейших возможностей языка C# 9 и .NET 5 вместе с подробным "закулисным" обсуждением, призванным расширить навыки критического мышления разработчиков, когда речь идет об их ремесле. Книга охватывает ASP.NET Core, Entity Framework Core и многое другое наряду с последними обновлениями унифицированной платформы .NET, начиная с улучшений показателей производительности настольных приложений Windows в .NET 5 и обновления инструментария XAML и заканчивая расширенным рассмотрением файлов данных и способов обработки данных. Все примеры кода были переписаны с учетом возможностей последнего выпуска C# 9.

Погрузитесь в книгу и выясните, почему она является лидером у разработчиков по всему миру. Сформируйте прочный фундамент в виде знания приемов объектно-ориентированного проектирования, атрибутов и рефлексии, обобщений и коллекций, а также множества более сложных тем, которые не раскрываются в других книгах (коды операций CIL, выпуск динамических сборок и т.д.). С помощью этой книги вы сможете уверенно использовать язык C# на практике и хорошо ориентироваться в мире .NET.

Основные темы книги

- Возможности языка C# 9 и обновления в записях, неизменяемых классах, средствах доступа только для инициализации, операторах верхнего уровня, сопоставлении с образцом и т.д.
- Начало работы с веб-приложениями и веб-службами ASP.NET Core
- Использование Entity Framework Core для построения реальных приложений, управляющих данными, с расширенным охватом нововведений этой версии
- Разработка приложений с помощью C# и современных инфраструктур для служб, веб-сети и интеллектуальных клиентов
- Философия, лежащая в основе .NET
- Новые средства .NET 5, включая однофайловые приложения, уменьшенные образы контейнеров, поддержку Windows ARM64 и многое другое
- Разработка настольных приложений Windows в .NET 5 с использованием Windows Presentation Foundation
- Улучшение показателей производительности благодаря обновлениям ASP.NET Core, Entity Framework Core и внутренних механизмов, таких как сборка мусора, `System.Text.Json` и оптимизация размера контейнера

Категория: языки программирования/C#
Предмет рассмотрения: C# 9
Уровень: для пользователей средней и высокой квалификации

ISBN 978-617-7987-81-8

Apress®
www.apress.com

Комп'ютерне видавництво
"Діалектика"
www.dialektika.com